

Нужная информация всегда под рукой!

Борис Карпов
Татьяна Баранова

Типы данных

Операторы
языка

Библиотечные
функции

Ключевые
слова



Специальный справочник

 ПИТЕР®

**Борис Карпов
Татьяна Баранова**

C++

Специальный справочник

**Санкт-Петербург
Москва • Харьков • Минск
2001**

 **ПИТЕР®**

Борис Карпов, Татьяна Баранова
С++: специальный справочник

Главный редактор
Заведующий редакцией
Руководитель проекта
Литературный редактор
Художник
Корректоры
Верстка

*В. Усманов
Е. Строганова
И. Корнеев
А. Жданов
Н. Биржаков
С. Беляева, А. Пученкина
А. Миронов*

ББК 32.973–018я22

УДК 681.3.06(03)

Карпов Б., Баранова Т.

К26 С++: специальный справочник — СПб: Питер, 2001. — 480 с.: ил.

ISBN 5–272–00076–5

В книге содержится обширный справочный материал по основным элементам языка, включающий в себя наиболее употребительные функции, операторы, директивы и классы, являющиеся ядром языка С++, не зависящим от аппаратной платформы или версии языка. Последовательно и подробно изложен процесс создания программ. Обзор способов решения различных задач представляется в виде сравнения возможностей двух языков — С и С++. Такая насыщенность информацией делает книгу полезной не только начинающим программистам, но и пользователям, активно работающим с С и С++.

© Б. Карпов, Т. Баранова, 2000

© Издательский дом «Питер», 2001

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 5–272–00076–5

Лицензия ИД № 01940 от 05.06.2000.

Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 95 3000 – книги и брошюры.

Подписано к печати 01.06.2001. Формат 84x108¹/₃₂. Усл. п. л. 25,2. Доп. тираж 5000 экз. Заказ № 574.

ЗАО «Питер Бук». 196105, Санкт-Петербург, ул. Благодатная, 67.

Отпечатано с готовых диапозитивов в ГИПК «Лениздат» (типография им. Володарского)
Министерства РФ по делам печати, телерадиовещания и средств массовых коммуникаций.
191023, Санкт-Петербург, наб. р. Фонтанки, 59.

Краткое содержание

Введение	17
-----------------------	-----------

Часть I. Справочник по основным элементам C++

Словарь терминов и понятий	28
---	-----------

Типы данных	84
--------------------------	-----------

Операторы	98
------------------------	-----------

Операторы приведения типов	116
---	------------

Ключевые слова	128
-----------------------------	------------

Директивы, макросы и операторы препроцессора	159
---	------------

Библиотечные функции	178
-----------------------------------	------------

Объекты и классы библиотеки ввода/вывода	261
---	------------

Часть II. Краткий курс программирования на C++

Основные особенности программирования на C++	292
---	------------

Указатели, массивы и строки	311
--	------------

Ввод и вывод	329
---------------------------	------------

Классы	341
---------------------	------------

Конструкторы	355
Перегрузка операторов	366
Наследование	389
Виртуальные функции	409

Часть III. Приложения

Отличия C и C++	426
Особенности ANSI C++	429
Стандартные исключения	435
Таблица ASCII-кодов	437
Список характерных задач в C++	439
Алфавитный указатель	465

Содержание

Введение	17
-----------------------	-----------

С и С++	18
Объекты	19
Инкапсуляция	19
Полиморфизм	21
Подход к типам в С++	21
Структура книги	22
Кому адресована эта книга	23
Принятые обозначения	24
От издательства	26

Часть I. Справочник по основным элементам С++

Словарь терминов и понятий	28
---	-----------

Абстрактные типы данных	28
Абстрактный класс	28
Абстракция данных	29
Агрегаты	29
Анонимные объединения	30
Аргумент	31
Ассоциативность	31
Базовый класс	32
Беззнаковые типы данных	32
Безопасная сборка программы с учетом типов	32
Битовое поле	33
Булевы значения	33
Виртуальная функция	34
Вложенные управляющие структуры и вложенные классы	35
Встраиваемая функция	35

Выражение	36
Главная функция	36
Данные-члены	37
Декорирование	37
Деструктор	37
Директива	38
Директива препроцессора	39
Дополнительный формат	39
Знакорасширение	40
Значения аргументов по умолчанию	41
Идентификатор	41
Именующее выражение	43
Инкапсуляция	43
Инструкция	44
Информация о типах во время выполнения программы	45
Исключение	45
Исполняемая инструкция	46
Класс	46
Класс памяти	47
Ключевое слово	47
Комментарии	47
Константа	48
Конструктор	49
Конструктор базового класса	50
Конструктор копирования	51
Конструктор по умолчанию	51
Косвенность	52
Литерал	52
Логические и битовые операторы	52
Макрос	53
Массив	54
Метка	54
Множественное наследование	55
Надкласс	56
Наследование	56
Область действия	56

Объект	57
Объектная ориентация	57
Объектно-ориентированное программирование	58
Объявление	58
Операнд	60
Оператор	60
Операторы присваивания	61
Определение	62
Параметр	62
Перегрузка операторов	63
Перегрузка функций	63
Переменная	64
Перечисление	64
Подкласс	65
Полиморфизм	65
Прагма	66
Приведение типа	66
Приоритет	67
Продвижение	68
Производный класс	68
Простейшие типы данных	69
Пространство имен	69
Прототип	70
Прототип функции	70
Пустая инструкция	70
Разыменованное	71
Реализация	72
Рекурсия	72
Сигнатура	73
Символ новой строки	73
Сложные типы данных	73
Создание экземпляров класса	74
Составные инструкции	74
Ссылка	74
Ссылка вперед	75
Строка	76
Строковый литерал	76

Текстовая строка	77
Указатель	77
Указатель на функцию	77
Управляющая последовательность	78
Управляющая структура	78
Условная компиляция	79
Файлы заголовков	79
Функции	80
Функции обратного вызова	80
Функции преобразования	81
Функция-член	81
Чистая виртуальная функция	82
Член	82
Шаблон	82
Экземпляр	83
Экземпляр класса	83
Типы данных	84
bool	88
char	89
unsigned char	91
signed char	91
int	92
unsigned int	93
short	94
unsigned short	94
long	94
unsigned long	95
float	95
double	96
long double	97
wchar_t	97
Операторы	98
Ключевые слова — операторы	103
Оператор sizeof	103

Оператор new	104
Оператор delete	105
Оператор typeid	105
Операторы присваивания	107
Поразрядные операторы	108
Оператор запятая	108
Декремент	109
Инкремент	109
Условный оператор	110
Логические операторы	110
Оператор деления по модулю	111
Операторы-указатели	111
Операторы-указатели на член объекта	112
Операторы отношения	113
Оператор определения области видимости	114

Операторы приведения типов 116

Оператор приведения типа старого стиля	117
Удаление модификатора const	118
Проверка типа во время выполнения программы	120
Приведение типа указателя	122
Преобразование типа между родственными объектами или указателями	124

Ключевые слова 128

asm	128
auto	128
break	128
case	128
catch	129
class	130
const	131
continue	132
default	132
do	132

else	133
enum	133
explicit	134
extern	135
for	135
friend	136
goto	137
if	137
inline	138
main	139
mutable	139
namespace	140
operator	141
private	142
protected	142
public	142
register	142
return	143
static	143
struct	146
switch	147
template	147
this	150
throw	150
try	151
typedef	153
union	154
using	155
virtual	155
void	156
volatile	156
while	157

Директивы, макросы и операторы препроцессора 159

Директивы	160
#define	160
#elif	164

#else	164
#endif	165
#error	165
#if	166
#ifdef	169
#ifndef	169
#include	170
#line	171
#pragma	172
#undef	173
Предопределенные макросы	173
__cplusplus	174
__DATE__	174
__FILE__	175
__LINE__	175
__TIME__	176
Операторы препроцессора	176
Библиотечные функции	178
Обзор библиотечных функций	178
Поддержка списков параметров переменной длины	181
Функции консольного ввода / вывода	183
Обзор	184
Чтение символов с клавиатуры	185
Чтение чисел с клавиатуры	186
Вывод символов на экран	186
Сравнение консольного и файлового ввода / вывода	187
Функции проверки категории символов	189
Функции работы с блоками памяти	191
Функции работы с датой и временем	193
Обзор	193
Использование структуры tm	195
Сложное формирование даты / времени	196
Функции работы со строками	197

Функции файлового ввода / вывода	201
Обзор	202
Чтение и запись текстовых файлов	205
Чтение и запись двоичных файлов	207
Сводный список функций	210
Объекты и классы библиотеки ввода/вывода	261
Краткий обзор библиотеки классов ввода / вывода	261
Организация ввода / вывода с помощью классов	263
Расширение оператора вывода	263
Расширение оператора ввода	265
Манипуляторы ввода / вывода	266
Флаги форматирования	268
Классы нового стиля	270
Обзор объектов и классов ввода / вывода C++	271
cerr	271
cin	272
clog	273
cout	273
fstream	274
ifstream	275
istream	279
istrstream	283
ofstream	283
ostream	286
ostrstream	288
strstream	288
 Часть II. Краткий курс программирования на C++	
Основные особенности программирования на C++	292
Первая программа на C++	292
Объявление переменных	294
Директива #include	296
Ввод данных	297
Операции, выражения и комментарии	298

Функции	300
Локальные, глобальные и другие переменные	302
Управляющие инструкции	304
Особые операции C++	306
Логические операции и операции над битами	308
Операции над полями бит	309
Указатели, массивы и строки	311
Указатели и передача данных	311
Передача данных по ссылке	313
Одномерные массивы	314
Многомерные массивы и их размещение в памяти компьютера	315
Инициализация массивов	318
Массивы и указатели	319
Строки	320
Указатели и динамическое распределение памяти	323
Использование стандартных функций malloc и free	324
Использование операторов new и delete	326
Ввод и вывод	329
Понятие потока	329
Операторы потоков для ввода/вывода	330
Управление форматом при работе с потоками	332
Ввод/вывод строк с помощью библиотеки stdio.h	333
Функции для работы с файлами в библиотеке stdio.h	335
Операторы потоков для работы с файлами	337
Операторы или функции?	340
Классы	341
Создание нового класса	341
Поведение класса	343

Объекты класса	345
Указатели на объекты	347
Достоинства классов	348
Конструкторы и деструкторы: рождение и смерть объектов	351
Определение встраиваемой функции	353
Структура как разновидность класса	354
Конструкторы	355
Перегрузка конструкторов	355
Конструктор по умолчанию	357
Конструктор копирования	357
Ссылки и конструктор копирования	359
Защита параметра от изменений	361
Примеры конструкторов	362
Вызов конструкторов и конверсия	364
Перегрузка операторов	366
Базовый синтаксис	366
Создание функций оператора сложения	367
Для чего нужны функции-друзья	369
Создание функции оператора присваивания	372
Указатель this и его использование при создании функций	374
Создание функции преобразования	375
Обобщение класса MyStr	377
Еще один пример перегрузки операторов	380
Обобщение определения операторов	382
Имя функции оператора	382
Бинарные операторы	382
Унарные операторы	383
Оператор присваивания	384
Присваивание переменных разных типов	384
Специальные операторы присваивания	385
Операторы инкремента и декремента	385
Оператор выделения элемента массива	386

Операторы new и delete	387
Оператор вызова функции	387
Ограничения на перегрузку операторов	388

Наследование 389

Дилемма программного обеспечения	389
Создание производного класса	390
Иерархия наследования	395
Создание класса без наследования	396
Наследование и оболочка объекта	399
Виды доступа: public, private и protected	400
Пример создания класса	401
Конструкторы базовых классов	404
Базовые классы и указатели	406

Виртуальные функции 409

Применение ключевого слова virtual	409
Программирование команд меню	413
Достоинства виртуальных функций	418
Чистые виртуальные функции и абстрактные классы	419
Определение виртуальной функции	420
Механизм вызова виртуальных функций	422

Часть III. Приложения

Отличия C и C++ 426

Особенности ANSI C++ 429

Новый стиль файлов заголовков	429
Операторы приведения типов ANSI	430
Шаблоны и обработка исключений	431
Новые ключевые слова	432
Область видимости в условной инструкции if	432
Перегрузка функций с перечисляемыми типами	433
Ссылки вперед для вложенных классов	434

Стандартные исключения	435
Таблица ASCII-кодов	437
Список характерных задач в C++	439
Алфавитный указатель	465

Введение

В начале компьютерной эры программисты были рабами вычислительных машин. Разработчики программного обеспечения должны были писать свои команды на единственном языке, который понимали компьютеры, — в двоичном коде, и программы выглядели как последовательность нулей и единиц. По мере того как время шло и алгоритмы усложнялись, программирование требовало все больше времени, а внесение изменений в программы и их модернизация становились практически невозможными. Так появились языки программирования высокого уровня: Фортран, Бейсик, Паскаль.

Требования к программам росли, времени для их написания отводилось все меньше, программистам надо было сосредоточиться на сложных алгоритмах, их эффективной реализации, не отвлекаясь на внутреннюю структуру компьютера. А тут еще проблемы переносимости программ на новые компьютеры с новыми возможностями... Был необходим новый подход — и он появился в виде объектно-ориентированного программирования.

Язык Simula, использовавшийся в 70–80-х годах в норвежских вооруженных силах, является одним из первых языков, основанных на понятии класс (класс — подмножество, состоящее из данных и связанных с ними функций). Примерно в то же время был утвержден стандарт нового языка программирования, получивший название С (Си) и обладавший большой мощностью, гибкостью и эффективностью. Достаточно сказать, что это был один из первых языков высокого уровня, позволявший работать с оборудованием, например, организовывать обмен данными между компьютерами.

В 1978 году были сделаны первые попытки объединить достоинства этих двух языков: так появился язык «С с клас-

сами». Лишь спустя несколько лет он стал тем C++, который так широко применяется сегодня.

Проходит время, меняются требования и подходы к программированию, и в соответствии с этими требованиями меняется язык. Сейчас существует множество различных платформ и версий C++. Среди них можно выделить наиболее часто используемые Microsoft C++ (DOS, WINDOWS) и Borland C++ (DOS, WINDOWS), Visual C++ (WINDOWS), GNU C++ (LINUX). В связи с этим можно говорить только о единстве ядра C++, в то время как интерфейс среды программирования и некоторые инструкции различаются для разных платформ и версий. В связи с этим в книге специально не рассматривается среда программирования, способы отладки и компиляции программы. При разработке примеров главным образом использовалась среда Borland C++ 4.5, которая, с нашей точки зрения, зарекомендовала себя как одна из самых надежных и может быть особенно привлекательна для начинающих программировать на C++.

С и C++

Язык C++ не требует обязательного применения объектов в программах. Это позволяет модернизировать ранее написанные и создавать новые программы, пользуясь практически синтаксисом С и выполняя лишь более строгие требования C++ к типам: наличие в начале программы прототипов всех функций, определенных пользователем, явного приведения типов для указателей на разные типы и некоторых других. Подробнее с различиями С и C++ можно ознакомиться в приложении А.

Однако архитектура современных операционных систем становится все более и более объектно-ориентированной. При работе в таких системах не обойтись без понимания и использования таких основополагающих понятий объектно-ориентированного программирования, как объекты, инкапсуляция и полиморфизм. Не стоит пугаться этих слов. Если внимательно посмотреть вокруг, весь мир со-

стоит из объектов, каждый из них в той или иной мере является «черным ящиком» и выполняет действия, называемые одним и тем же словом, совершенно по-разному. Более подробно эти понятия будут рассмотрены в главах «Классы», «Конструкторы» и «Перегрузка операторов». А пока небольшая прелюдия.

Объекты

Одним из центральных понятий в объектно-ориентированном программировании (ООП) является объект. Чтобы пояснить понятие объекта, его свойств и событий в ООП, можно использовать сравнение с окружающим нас миром. Мир состоит из живых и неживых объектов, определяемых некоторыми свойствами и действиями. Чтобы установить, изменить или использовать те или иные свойства определенного объекта, необходимо выполнить соответствующие действия.

Примером объекта может служить точка на экране компьютера. Определяющими ее свойствами являются координаты и цвет. Точку можно нарисовать, стереть, скопировать с новыми координатами, изменить или узнать ее цвет, то есть выполнить над ней некоторые действия.

Каждый объект реализуется на основе класса — структуры (группы) данных, которая может иметь связанные с ней встроенные функции.

В отличие от традиционного подхода к программированию, где данные и обрабатывающие их программы (команды компьютера) существуют сами по себе и не связаны идеологически, объект содержит то и другое. При этом данные называются свойствами, а способы их обработки — функциями или методами.

Инкапсуляция

Одним из основополагающих принципов объектно-ориентированного программирования является независимость внутренней структуры объектов друг от друга («идея черного ящика»).

С помощью заранее определенных графических точек мы можем построить прямые (точнее говоря, отрезки прямых), эллипсы, дуги и также определить их как объекты, но уже с другими свойствами. Например, свойствами объекта «прямая» будут координаты начала и конца, ее цвет, а функциями — нарисовать прямую, изменить цвет, скопировать.

Такой подход имеет преимущества при изменении уже готовых программ. Например, если при создании функции для объекта «прямая» используется объект «графическая точка», то при различных внешних изменениях (предположим, при установке иного способа передачи цвета точки) будет достаточно изменить только этот объект, не затрагивая прочие, основанные на нем (то есть объекты «прямая», «эллипс», «дуга» и т. д.). Та часть программы, которая была написана с использованием этих объектов, останется без изменений.

Такая возможность имеет особенно важное значение при разработке большой системы, в которой участвует несколько десятков, а то и сотен программистов. При объектно-ориентированном подходе изменение одного объекта не потребует никаких изменений в других объектах.

В свете всего сказанного становится ясным еще один важный принцип ООП — прозрачность объектного интерфейса (данных и методов объекта, доступных внешнему миру). Если свойства и функции объекта определены правильно, становится возможной безболезненная замена внутреннего содержания любого из них с полным сохранением работоспособности системы.

Итак, независимость внутренней структуры объектов и прозрачность их интерфейса — два краеугольных камня объектно-ориентированного программирования.



ПРИМЕЧАНИЕ На первом этапе изучения C++ можно использовать термины объект и класс как взаимозаменяемые. Но отличие между ними существенное: класс — это тип, а объект — представитель или экземпляр (переменная) этого типа. (См. также главу «Классы».)

Полиморфизм

Одним из принципов объектно-ориентированной системы является максимально возможная независимость каждого объекта. Но что делать, если одна из функций объекта должна изменяться в зависимости от ситуации?

В качестве примера можно привести все тот же объект «графическая точка» и изменение ее формы на экране для различных случаев. Для объектов «ось координат» и «график кривой» основная программа рисования графика будет использовать стандартную функцию «нарисовать точку», а форма точки на экране будет определяться специальными версиями функции «нарисовать точку» в объектах «ось координат» и «график кривой».

В теории ООП это свойство называется полиморфизмом (в переводе с греческого — «много форм»).

Подобный метод реализуется и в языке C посредством функций обратного вызова. Однако полиморфизм является более надежным, удобным и наглядным механизмом для замены стандартной функции ее конкретной реализацией.



ПРИМЕЧАНИЕ Более подробно полиморфизм, наследственная иерархия и виртуальные функции описаны в главах «Наследование» и «Виртуальные функции».

Подход к типам в C++

Объектно-ориентированный подход является наиболее интересной особенностью C++. Однако не менее важным следует считать более строгий по сравнению с C подход в C++ к использованию типов. Это делает программы на C++ более устойчивыми к ошибкам во время выполнения программы, независимо от того, используются методы ООП или нет.

К преимуществам C++ можно отнести и возможность повторного использования имен. В частности, одно и то же

имя функции может быть определено для аргументов разного типа (в С для вычисления абсолютной величины целого числа и числа с плавающей точкой приходилось пользоваться разными функциями).

В действительности эти функции совершенно различны, имеют разные определения и объявления, и чтобы различить их, С++ использует информацию в списке аргументов.

(См. также главы «Классы», «Конструкторы» и «Перегрузка операторов».)

Структура книги

В книге реализованы две глобальные цели, всегда возникающие при создании книги для изучения любого языка программирования. С одной стороны — это справочник (часть I), который содержит наиболее полную информацию об основных элементах С++, а с другой — учебник (часть II), в котором последовательно рассказывается о принципах создания пользовательских программ. Особенностью книги является способ предоставления информации. Справочные материалы собраны в тематические разделы, из которых можно быстро вычленить элемент, необходимый для конкретной ситуации, а учебные материалы, опираясь на справочную информацию, предоставляют возможность осваивать язык активно — в процессе разбора программ-примеров.

Каждой обсуждаемой теме в части I, «Справочник по основным элементам С++», посвящена отдельная глава, которая, в свою очередь, разбита на разделы. Информация излагается, насколько это возможно, последовательно, то есть пользователь, начинающий осваивать С++, сначала получит информацию о типах данных, а уже затем о ключевых словах и стандартных функциях языка. Для удобства поиска необходимой информации разделы в некоторых главах части I расположены в алфавитном порядке. Например, главы, посвященные ключевым словам и функциям, выстроены в соответствии с английским алфавитом,

а глава, посвященная основным терминам и понятиям языка, — в соответствии с русским.

Часть II, «Краткий курс программирования на C++», имеет структуру, подобную структуре части I, но принцип, по которому она построена, иной. Каждая глава части II посвящена обсуждению конкретной задачи, название которой вынесено в заголовок главы. Если задача может быть решена путем последовательного решения более мелких задач, то каждой такой задаче посвящен отдельный раздел. Все главы части II, в отличие от части I, связаны воедино, поэтому (особенно для начинающих пользователей) рекомендуется знакомиться с материалом последовательно.

Часть III, «Приложения», содержит материал, который может быть интересен при разработке конкретных программ. Так, приложения А и Б будут полезны пользователям, переходящим с языка С на C++ или решившим адаптировать свои программы к требованиям стандарта ANSI C++. В приложении В описываются стандартные исключения C++, использование которых позволит избежать непредсказуемых ситуаций при работе программы, способных повлечь ее аварийное завершение или нарушения в работе операционной системы. В приложении Г собраны ASCII-коды символов, а приложение Д содержит таблицу, в которой представлены типовые задачи, возникающие при разработке программ, и стандартные элементы языка, позволяющие эти задачи решить. Это приложение пригодится начинающим пользователям, у которых еще нет четкого представления обо всех возможностях C++.

Кому адресована эта книга

Основная цель книги — помочь пользователю решить стоящую перед ним задачу, предоставив ему наиболее полную информацию за минимум времени. Поэтому, с одной стороны, информация представлена предельно сжато, а с другой, она достаточна для понимания сущности исполняемого процесса. Кроме общей информации все описываемые элементы языка содержат подробные примеры. Для

целостности картины многие примеры являются «сквозными», то есть процесс их создания и модификации рассматривается на протяжении нескольких глав. Таким образом, можно разобраться в основополагающих принципах C++, а также узнать о наиболее быстрых способах выполнения традиционных задач.

Опытным программистам, в первую очередь, может быть полезен материал части I, «Справочник по основным элементам C++», которая, благодаря удобной структуре и многочисленным перекрестным ссылкам, позволяет быстро найти синтаксис необходимого элемента, а также рекомендации по его использованию в той или иной ситуации. Для начинающих последовательное изучение материала части I может оказаться непосильной задачей. Поэтому им рекомендуется после ознакомления с первыми главами переходить к части II, «Краткий курс программирования на C++», а к части I обращаться по мере необходимости.

Нужно заметить, что объем и цель данного издания не позволяют более детально рассмотреть возможности C++, ориентированные на конкретные платформы и версии. Читателям, желающим получить более полную информацию по этим вопросам, можно рекомендовать книгу П. Франка «C++. Учебный курс», вышедшую в издательстве «Питер».

Миниатюрный размер книги в сочетании с представленной в ней обширной информацией делает ее совершенно незаменимой на столе как профессионального программиста, так и пользователя, начинающего осваивать C++.

Принятые обозначения

Для удобства изложения и восприятия материала в этой книге используются некоторые специальные обозначения.

Новые термины, которые встречаются в тексте, выделены *курсивом*. В главе «Словарь терминов и понятий» все эти термины с их подробным описанием собраны вместе. Если вы забыли значение какого-нибудь термина, можете об-

ращаться к этой главе. Фрагменты кода обозначаются моноширинным шрифтом. Для названий файлов и сообщений, выдаваемых компилятором, также используется специальный шрифт.

Иногда, в синтаксисе стандартных элементов языка для лучшей наглядности, используется русская терминология. Такие слова заключены в угловые скобки. При включении слова в текст программы его следует опустить вместе со скобками, а на его место вставить необходимые аргументы или параметры. Рассмотрим следующий фрагмент:

```
{<тело функции>}
```

В тексте программы текст <тело функции> необходимо заменить реальными инструкциями, например, такой:

```
{  
    return String;  
}
```

Если названия нескольких клавиш разделены знаком плюс, например Ctrl+V, вам нужно, удерживая нажатой первую клавишу, нажать вторую.

Во многих разделах имеются рубрики «Внимание», «Примечание» и «Совет», призванные привлечь внимание читателя. Каждая рубрика имеет свою специфику.



ВНИМАНИЕ Будьте осторожны, когда встретите эту рубрику. Гораздо лучше учиться на ошибках других, чем совершать ошибки самому.



ПРИМЕЧАНИЕ В эту рубрику вошли некоторые рассуждения, связанные с рассматриваемой темой, но лежащие вне контекста текущего раздела. Помните, в примечаниях часто содержатся крупницы мудрости.



СОВЕТ Здесь содержатся советы тех, кто действительно знает, как работать с программами. В них вы найдете информацию о том, как сэкономить время при выполнении определенных задач или выбрать кратчайший путь их решения.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter-press.ru (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на Web-сайте издательства <http://www.piter-press.ru>.

Часть I

**Справочник по основным
элементам C++**

Словарь терминов и понятий

Абстрактные типы данных

В теории объектно-ориентированного программирования одним из самых сложных понятий является понятие абстрактных типов данных (abstract data types). Абстрактный тип данных — это тип данных, для которого детали реализации не определены: они могут появиться позднее или вообще быть скрытыми. Эта концепция является ядром объектного программирования, в котором сначала определяется соотношение между объектами и классами, предназначенными для работы, а конкретные детали прорабатываются позднее. С практической точки зрения эта концепция не так важна, за исключением одного пункта: языки типа C++ предоставляют возможность для скрытия деталей реализации, поэтому тип данных определяется его поведением, а не способом построения.

Абстрактный класс

Абстрактный класс (abstract class) — это класс, непосредственное создание объектов которого невозможно, но который используется как базовый для построения производных классов.

Например, можно определить абстрактный класс CShape (фигура), который определяет общие свойства геометрической фигуры. Непосредственное объявление объектов этого класса невозможно, но он может служить базовым для таких классов, как CSquare (квадрат), CCircle (круг), CTriangle (треугольник) и тому подобных. В лексиконе программистов абстрактный класс — это любой класс с одной или более чистой виртуальной функцией. Чистая виртуальная функция не имеет определения в том классе, в котором она объявлена, и реализуется в производных классах (см. раздел «Чистая виртуальная функция»).

Абстракция данных

Абстракция данных (data abstraction) — искусство определения типов в терминах того, что они должны делать, но не в терминах их внутренней структуры (см. также раздел «Абстрактные типы данных»).

Агрегаты

Агрегаты (aggregates) — это списки констант, используемых для инициализации сложных типов данных. Они могут быть представлены в двух формах. Первая форма — это набор. Набор состоит из одного или более числа членов, каждый из которых может быть простым значением или другим агрегатом (таким как строка). В определении агрегата фигурные скобки являются обязательными элементами, между которыми можно включить любое число членов:

```
{ член [, член]... }
```

Вторая форма представления агрегата — это строка символов, которая используется для инициализации массивов типа `char`.

Синтаксис:

"Текст"

Например, в следующей инструкции инициализируется массив указателей типа `char*`:

```
char *cats[3] = {"Bill", "CB", "Purrly"};
```

В этом случае каждый из трех указателей инициализируется указателем на свою строку символов. А в следующем примере инициализируются не все указатели:

```
char *cats[5] = {"Bill", "CB", "Purrly"};
```

В этом случае массив `cats` состоит из пяти указателей типа `char*`, но только три первых инициализируются, а последние два не инициализируются и их значения остаются равными `NULL`. При использовании вложенности во время инициализации сложных структур каждый вложенный

член инициализирует соответствующую логическую часть. Например, следующее определение заполняет три строки двумерного массива `matrix1` размером 3×6 , однако инициализируется только первая половина каждой строки (то есть первые три значения из шести). Неинициализированным элементам по умолчанию присваивается значение ноль (0).

```
int matrix1[3][6] = {{1,2,3}, {4,5,6}, {7,8,9}};
```

Агрегаты играют особую роль при инициализации массивов, длина которых не определяется в явном виде. В этом случае число членов агрегата определяет размер отводимого для массива пространства. Например, следующее определение размещает массив ровно из шести элементов:

```
int lotto_numbers[] = { 1, 5, 21, 27, 33, 40 };
```

Анонимные объединения

Анонимные объединения (anonymous union) — это объединения, имена которых не заданы в явном виде и определяются по имени используемого члена объединения. Стандарт ANSI C++ описывает такие объединения. Рассмотрим такой пример:

```
struct databag
{
    char fancy_name[20];
    union
    {
        char strdata[20];
        double floatdata;
    }
} bag1;
```

При таком определении можно использовать члены `bag1.strdata` и `bag1.floatdata` без указания имени объединения. Существует только одно принципиальное ограничение: использование анонимного объединения не должно создавать конфликта имен.

В приведенном выше примере конфликта имен нет, так как `fancy_name` отличается от `strdata` и `floatdata` (см. также раздел «union» в главе «Ключевые слова»).

Аргумент

Аргумент (argument) или параметр (parameter) — это значение, передаваемое в функцию или шаблон. Например, в случае вызова функции вычисления факториала `fact (4)`, возвращаемым значением будет число $4!$ (4 факториал), `fact (5)` возвращает значение $5!$ (5 факториал). В этом случае числа 4 и 5 являются аргументами или параметрами. В определении функции `fact` (см. разделы «Определение» и «Рекурсия») параметр может быть объявлен как переменная целого типа `n`:

```
long fact(int n) { // ... }
```

В некоторых книгах по программированию используются термины параметр и аргумент для различия определения параметра (как `n` в данном случае) и значения, передаваемого при вызове функции (4 и 5 в данном случае). Другие книги используют термины формальный и фактический параметры, соответственно, для тех же целей. В этой книге для простоты в обоих случаях, как правило, используется термин параметр.

Ассоциативность

Ассоциативность (associativity) операторов определяет порядок выполнения операторов в сложном выражении, если они имеют одинаковые приоритеты (см. раздел «Приоритет»). Так, сложение (+) и вычитание (-) имеют одинаковый приоритет, при этом в случае последовательного выполнения этих операторов их ассоциативность определяется слева направо. Например, выражение

```
amount = 10 - 5 + n
```

эквивалентно следующему:

```
amount = (10 - 5) + n // amount = 5 + n
```

что, конечно, отличается от такой интерпретации:

```
amount = 10 - (5 + n) // amount = 5 - n
```

Все операторы C++ ассоциативны слева направо, за исключением операторов присваивания, унарных операторов и

условного оператора (`?:`). В случае сомнения можно проверить ассоциативность и приоритет операторов по табл. 3 и 4 (см. главу «Операторы»).

Базовый класс

Понятие базового класса (base class) перекликается с понятием производного класса (см. также раздел «Создание производного класса» в главе «Наследование»). Все члены базового класса автоматически становятся членами производного класса. Например, если класс В является базовым классом для класса С, то члены класса В автоматически будут являться членами класса С. Это отношение является примером наследования в С++. Иногда базовый класс называется суперклассом, классом-предком или родительским классом. (См. также раздел «Производный класс».)

Беззнаковые типы данных

В С и С++ поддерживаются как знаковые, так и беззнаковые типы данных (unsigned data types). Все целые по умолчанию подразумеваются знаковыми, а для объявления беззнакового целого необходимо использовать ключевое слово `unsigned`. Беззнаковые целые очень просты для понимания. Все битовые комбинации представляют либо значение ноль (0), либо положительные числа. Недостатком беззнакового представления является то, что с его помощью невозможно представить отрицательные числа. Однако, жертвуя отрицательными значениями, можно получить вдвое больший диапазон положительных значений. Например, диапазон знаковых двухбайтных целых составляет от -32 768 до 32 767, а диапазон беззнаковых двухбайтных целых — от 0 до 65 535 (см. также раздел «Дополнительный формат»).

Безопасная сборка программы с учетом типов

В отличие от С в С++ поддерживается безопасная сборка программы с учетом типов (type-safe linkage). Таким об-

разом, удастся не допустить сборки программы, в которой, например, в одном модуле переменная объявлена как `short`, а в другом — как `long`. Эта защита основывается на схеме декорирования имен (см. также раздел «Декорирование»).

Битовое поле

Битовое поле (`bit field`) — это специальный тип данных в C/C++, который является последовательностью бит в переменной типа `int` или `unsigned int`. Такое поле может состоять из одного бита, хотя более распространены поля из нескольких бит. При использовании в программе битовых полей компилятор создает команды для извлечения, обработки и сохранения битовых значений. Несмотря на то что конечный результат будет таким же, как при поразрядных операциях (см. также раздел «Логические операции и операции над битами» в главе «Основные особенности программирования на C++»), у битовых полей есть два преимущества, одно из которых заключается в легкости написания и восприятия текста программы, а второе — в меньшем числе строк исходного текста для описания одних и тех же действий (см. также раздел «Операции над полями бит» в главе «Основные особенности программирования на C++»).

Булевы значения

Булево выражение — это выражение, которое может принимать одно из двух значений: `true` или `false`. Большинство типов данных имеют диапазон значений существенно больше, чем два значения. Если в программе есть проблемы с памятью, то использование однобитных полей для хранения булевых значений (`Boolean values`) наиболее эффективно (см. раздел «Битовое поле»). Так как доступ к отдельным битам требует лишних команд в программе, то в большинстве компиляторов по умолчанию для представления булевых переменных используется целое. В ранних версиях C++ нет отдельного булевого типа данных: значение ноль (0) рассматривается как `false`, а любое не-

нулевое значение — как `true`. Поэтому следует отличать логические (`&&`, `||`, `!`) и поразрядные операторы (`&`, `|`, `~`). Поразрядное `6 & 9` дает результат, равный 0 (эквивалентно `false`), в то время как логическое `6 && 9` — `true`. В стандарте ANSI C++ предлагается другой путь — использовать новый тип данных `bool`. Любое ненулевое значение, присваиваемое булевой переменной, автоматически переводится в значение единица. (См. также главу «Типы данных».)

Логические значения `true` и `false` названы булевыми по имени Георга Була (Georg Boole), родоначальника современной символической логики.

Виртуальная функция

Виртуальная функция (virtual function) — это функция, которая может объявляться различными способами в различных классах и которая правильно выполняется, как только появляется доступ к определяющему ее классу. Конкретная версия функции выбирается даже в том случае, если объект становится доступным через указатель на базовый класс или через массив объектов различных типов.

Можно представить себе, что каждый объект «знает», как выполнить предназначенную для него виртуальную функцию. Таким образом, реализуется поддержка концепции объектов как автономных единиц, которые могут действовать независимо, подобно клеткам в живом организме. Более того, новые типы объектов могут встраиваться в существующие рамки, принося с собой указатели на свои собственные реализации виртуальных функций. Хотя допускается перегружать функции, которые не объявлены виртуальными, делать это приходится на свой страх и риск. Общим правилом должно стать следующее: любая функция-член, которая может быть перегружена в производном классе, должна объявляться виртуальной в базовом классе.

Виртуальные функции тесно связаны с объектно-ориентированной концепцией полиморфизма (см. главу «Виртуальные функции»). Объявление виртуальных функций

производится с использованием ключевого слова `virtual` (см. также раздел «`virtual`» в главе «Ключевые слова»).

Вложенные управляющие структуры и вложенные классы

Вообще, вложенность — это размещение одного в другом. Таким образом, вложенная управляющая структура (`nested control structure`) — это инструкция, вложенная внутри другой. Такая возможность вытекает из синтаксиса: в инструкцию может быть вложена другая инструкция, но управляющая структура сама является инструкцией, следовательно, инструкция `if`, например, может содержать внутри себя инструкцию `while`, которая, в свою очередь, может содержать инструкцию `if`, и т. д. Инструкции цикла могут вкладываться друг в друга бесконечное число раз. Язык C++ также поддерживает вложенные классы (`nested classes`). На вложенный класс может ссылаться заключающий его класс, но не наоборот. Тем не менее стандарт ANSI C++ допускает ссылку из вложенного класса на внешний с предварительным использованием ссылки вперед (см. также раздел «Ссылка вперед»). Более подробно с этим можно познакомиться в приложении, «Особенности ANSI C++».

Встраиваемая функция

Когда компилятор встречает вызов встраиваемой функции (`inline function`), он просто вставляет определение такой функции в месте ее вызова. Встраиваемые функции подобны макросам в том смысле, что они не исполняются стандартным путем, то есть передачей управления на точку входа в функцию. Тем не менее в отличие от макросов, встраиваемые функции подчиняются синтаксису обычных функций. Согласно ANSI C++ существует два пути создания встраиваемых функций:

- в случае функции-члена класса встраиваемую функцию определяют внутри объявления класса, например:


```
class CHorse
{
    MyStr name;
    CHorse (MyStr nm)
    { name = nm; }
};
```

- ✱ встраиваемую функцию можно также получить при использовании в ее определении ключевого слова `inline`:

```
inline double cube_it(double x)
{return x * x * x;}
```

Кроме этого, некоторые компиляторы могут создавать встраиваемый код сами, определив, что именно этот вариант является оптимальным. (См. также главу «Классы» и раздел «`inline`» в главе «Ключевые слова».)

Выражение

Говоря общими словами, выражение (`expression`) в языках С и С++ — это то, что приводит к некоторому значению. Например, как простая переменная, так и формула $(x \cdot n + 1) / y^{2k}$ являются выражениями. Уникальным свойством С/С++ является то, что любое выражение можно превратить в инструкцию, добавив после него точку с запятой (;). Более того, операции присваивания также являются выражениями, так как они возвращают те значения, которые присваиваются. Все последующие примеры являются правильными выражениями, превращенными в инструкции добавлением точки с запятой:

```
x;
y = x;
y = x = z = 0;
y = ( x * n + 1 ) / y2k;
func1();
y = func1();
```

Главная функция

В каждой программе есть специальная функция `main`, которая получает управление при входе в программу и на-

зывается главной (`main`). Когда она завершается (инструкцией `return`, функцией `exit()` или по концу функции), нормально завершается и вся программа. Функция `main` должна определяться только для выполняемых программ, но не для динамически связываемых библиотек (DLL). (См. также раздел «`main`» в главе «Ключевые слова».)

Данные-члены

Данные-члены (`data members`) — переменные в определении класса. С их помощью определяют поля данных для каждого объекта класса. Существует одно исключение из этого правила: данные-члены, объявленные с модификатором `static`, не имеют соответствующих полей в каждом экземпляре. Наоборот, эти поля создаются только один раз для всех объектов класса, а значит, разделяются всеми объектами этого класса.

Декорирование

Декорирование (`decorating`) — это механизм, используемый компилятором C++ для сохранения информации о типе переменной вместе с ее именем. Например, если присвоить переменной типа `int` имя `Charley`, то компилятор поместит в выходной файл (объектный или выполняемый) не просто имя `Charley`, а добавит к этому имени дополнительные символы так, что потом эта переменная неявно будет определена как `Charley_as_int`. Существует две основных причины для декорирования. Первая — обеспечение безопасной сборки исполняемой программы. Так как имена переменных и функций в объектном файле декорированы, то при сборке программы переменные `Charley` типа `int` и `Charley` типа `float` будут различаться. Вторая причина — поддержка перегрузки функций. Следует отметить, что схема декорирования существенно различается в различных компиляторах и платформах.

Деструктор

Деструктор (`destructor`) — противоположность конструктора (см. раздел «Конструктор»). Если конструктор вы-

зывается при создании объекта, то деструктор вызывается для удаления объекта из памяти. Обычно нет необходимости определять деструктор для каждого класса в явном виде, так как он вызывается автоматически, например, при использовании оператора `delete`. Однако существуют ситуации, когда нужно аккуратное уничтожение объекта, например при закрытии файлов, связанных с объектом. В отличие от конструктора деструктор может быть только один для каждого класса. Пример записи деструктора:

```
~C();
```

(См. также главы «Классы» и «Конструкторы».)

Директива

Директива препроцессора (preprocessor directive), или просто директива (directive), — это специальная команда компилятора, которая исполняется перед компиляцией программы и ее запуском на выполнение. В отличие от инструкции директива не имеет никакого отношения к тому, что делается при выполнении программы, то есть директивы никак не изменяют время и ход выполнения программы. Одна из наиболее полезных директив — директива `#define`, которая, например, позволяет создать символическую константу:

```
#define PI 3.14159265
```

В разделе «Директивы, макросы и инструкции препроцессора» приведен полный список директив по стандарту ANSI вместе с их описанием.

При компиляции программы ее текст сначала обрабатывается препроцессором. Во время этого шага компилятор читает текст программы и изменяет его в соответствии с указанными директивами препроцессора. Изменения касаются только памяти, сам текст, конечно, не изменяется. Например, директива `#define` определяет подстановки, а директива `#include` вставляет на свое место указанный в этой директиве файл. Обработанный таким образом текст компилятор сохраняет в отдельном временном файле (по

умолчанию его не видно, однако можно указать компилятору на необходимость его сохранения). Такая обработка позволяет языкам С и С++ поддерживать широкий спектр мощных директив. Среди прочих стоит отметить возможность условной компиляции, что позволяет иметь несколько версий программы без необходимости сохранять несколько вариантов текстов. Другой приятной возможностью является вынесение общих определений констант, прототипов и т. д. в отдельный файл. После его включения можно быть уверенным, что все эти определения одинаковы во всех текстах программы. Использование препроцессорной обработки не приводит к увеличению ни занимаемой программой памяти, ни времени ее исполнения (возможно, немного увеличится время компиляции).

Директива препроцессора

(См. раздел «Директива».)

Дополнительный формат

Современные мини-компьютеры, так же как и персональные компьютеры, используют дополнительный формат (two's complement format) для представления отрицательных значений в знаковых целых значениях. В этом формате отрицательное число получается после выполнения следующих двух шагов.

1. Получают побитовую инверсию соответствующего положительного числа (например, для числа -1 значение 00000001 заменяется на 11111110).
2. Добавляется 1 (в нашем примере значение 11111110 заменяется на 11111111).

Одним из следствий такого представления является то, что число -1 изображается всеми единицами для целого любого размера. Другое следствие заключается в том, что единица в самом левом разряде означает отрицательное число. Следовательно, в случае знакового целого верхняя

половина диапазона битовых значений используется для представления отрицательных чисел. В двухбайтном целом с нулевым самым левым битом могут быть представлены числа от 0 до 32 767. А при значении бита в крайней левой позиции равном 1 будут представлены числа от -32 768 до -1.

В большинстве случаев способ представления целых программистом может не учитываться. Компилятор обычно генерирует подходящие инструкции процессора для обработки каждого типа данных. Таким образом, для работы со знаковыми и беззнаковыми целыми инструкции обычно одинаковы. Однако бывают ситуации, когда приходится создавать разные инструкции для целых знаковых и беззнаковых. Это необходимо при присваивании целого меньшего (в битах) размера целому большего (в битах) размера. В этом случае для беззнакового целого необходимо старшие разряды заполнить нулевыми битами, а для знакового целого необходимо добавить знаковый разряд. (См. также раздел «Знакорасширение».)

Знакорасширение

Знакорасширение (sign extension) — это механизм, используемый при присваивании целого знакового значения «большому» значению (см. раздел «Дополнительный формат»). Эту тонкость, зависящую от типа компьютера, можно совершенно спокойно проигнорировать, если не «смешивать» знаковые и беззнаковые данные различных размеров. В этих случаях необходимо уделять достаточно много внимания эффектам знакорасширения или расширения нулями. Например, если в однобайтном знаковом целом содержится значение -1, а в беззнаковом — значение 255, то оба они имеют одинаковый внутренний формат — все единицы (двоичный формат: 11111111), но их поведение различно в случае присваивания целой переменной. Для более спокойной жизни лучше использовать явное приведение значения к типу `signed` или `unsigned` перед присваиванием «большому» значению.

Значения аргументов по умолчанию

Каждый параметр в определении функции может иметь значение по умолчанию. Если при вызове функции этот параметр не задается, то используется значение аргумента по умолчанию (default argument value). Рассмотрим пример:

```
void set_vars( char *name, int amt = 0 );
```

Теперь, если написать вызов функции в виде:

```
set_vars("Joe Schmoe");,
```

то функция получит значение параметра `amt`, равное 0. А в случае вызова

```
set_vars("Joe Schmoe", 5);
```

переменная `amt` будет иметь значение 5. Существуют некоторые ограничения на использование значений аргументов по умолчанию. Во-первых, все параметры, имеющие значения по умолчанию, должны следовать после всех параметров, не имеющих такого значения. Другими словами, все параметры, которые могут не указываться при вызове функции, должны располагаться в конце списка аргументов. Это ограничение вызвано тем обстоятельством, что компилятор обрабатывает список аргументов слева направо. Сначала он обрабатывает весь имеющийся список параметров, а затем, обнаружив нехватку параметров, пытается дополнить список до нужной длины на основе значений по умолчанию. Во-вторых, существует ограничение для перегружаемых функций. Если функция перегружается, то сигнатуры (см. также разделы «Перегрузка функций» и «Сигнатура») должны отличаться, по крайней мере, одним параметром, не имеющим значения по умолчанию.

Идентификатор

Идентификатор (identifier) — это имя, определенное в программе. Любое слово в программе (за исключением

текста в строках или комментариях) является либо ключевым словом, либо идентификатором. Идентификаторы — это все имена, объявленные в программе, они могут быть именами переменных, типов, классов, макросов и функций. Кроме того, все имена, объявленные в библиотеке, также являются идентификаторами (а не ключевыми словами). Идентификаторы в C++, так же как и в C, формируются по следующим правилам:

- идентификатор не может совпадать с ключевым словом;
- каждый символ в имени идентификатора должен быть буквой (прописные и строчные буквы различаются), цифрой от 0 до 9 или символом подчеркивания (_);
- первый символ идентификатора не может быть цифрой.

При создании новых идентификаторов рекомендуется соблюдать еще некоторые правила:

- хотя никто не запрещает начинать имя идентификатора с символа подчеркивания, лучше этого не делать, так как подобные имена используются стандартной библиотекой C++ для служебных надобностей;
- старайтесь избегать имен идентификаторов, отличающихся друг от друга только регистром букв (типа `MrBig` и `mrBIG`). Компилятор поймет введенные идентификаторы, однако не каждый построитель исполняемой программы различает регистр букв, что может привести к конфликтам.

В процессе разработки программы на C++ желательно выработать систему написания идентификаторов (в части использования букв верхнего и нижнего регистра и т. п.). Наиболее известна так называемая венгерская схема, придуманная Чарльзом Симони (Charles Simonyi), одним из руководителей Microsoft, которая заключается в написании идентификатора, максимально полно отражающего его смысл. При этом в начале идентификатора сокращенно указывается его тип, например:


```
long *lpw; //рабочая переменная, являющаяся  
          //указателем на значение типа long  
int iCyclesNumber; //число циклов типа int
```

При другой схеме записи все идентификаторы, относящиеся к некоторому разделу, помечаются какой-то группой символов, например: NtAddAtom, NtGlobalFlag. В третьем варианте в качестве идентификатора записывается целая фраза, а вместо пробелов используются символы подчеркивания: set_user_defined_cursor, reset_system_cursor.

Именуемое выражение

Именуемое выражение (lvalue) — это выражение, которое можно использовать в левой части присваивания. Переменная может являться именуемым выражением, а константа нет. В общем, любое выражение, которое имеет адрес в памяти, является именуемым выражением. Например, если определить массив numbers, то сам адрес массива numbers к именуемым выражениям не относится, так как он постоянен и не занимает места в памяти, но numbers[i] уже является именуемым выражением. Указатели также относятся к именуемым выражениям. Если p определен как указатель, то p и *p — именуемые выражения. Кроме констант наиболее очевидным случаем выражения, которое не является именуемым, служит сложное выражение типа x+y. Несмотря на то что и x, и y имеют каждый свой адрес в памяти, выражение x+y адреса не имеет. Разыменованное выражение из указателя также является именуемым, так как для него есть адрес в памяти. Так, выражение p + i + 2 к именуемым выражениям не относится, а *(p + i + 2) — именуемое выражение.

Инкапсуляция

Инкапсуляция (encapsulation) предназначена для защиты «внутренностей» реализации. Хорошим стилем в объектно-ориентированном программировании являются сокры-

тие структуры внутренних данных от внешнего мира и разрешение взаимодействия только через допустимые «каналы» связи, что исключает возможность неправильных действий со стороны «внешнего» мира. В языке C++ это обеспечивается объявлением некоторых (или всех) членов класса закрытыми или защищенными (см. также раздел «private» в главе «Ключевые слова»). Инкапсуляция — один из трех столпов объектно-ориентированного программирования (два других — наследование и полиморфизм). Среди этих столпов инкапсуляция, возможно, самый главный, особенно для малых и средних по размеру проектов. Важность инкапсуляции заключается в том, что каждый класс (и, следовательно, объект) может изменяться или оптимизироваться в процессе разработки или даже после ее завершения без нарушения работоспособности остальных частей программы. Понятие инкапсуляции тесно связано с понятием абстракции. Грубо говоря, оба они означают одно и то же — сокрытие деталей. В главе «Классы» преимущества инкапсуляции объясняются на подробном примере.

Инструкция

Наряду с выражением инструкция (statement) является базовым элементом программы. Соотношение между этими понятиями достаточно понятно: выражение превращается в инструкцию путем добавления после него символа точки с запятой (;). Сами инструкции включают в себя управляющие структуры, такие как if, while и for. Синтаксис инструкции в языках C/C++ является рекурсивным, следовательно, инструкции можно конструировать из более мелких инструкций, образуя составные инструкции.

В C и C++ нет специальных инструкций присваивания — они являются частным случаем выражения, такого как $x=y$; (заканчивается точкой с запятой). Инструкция присваивания в таком случае имеет синтаксис:

выражение;

Программа на C++ — это последовательность директив, объявлений и определений функций. В отличие от программы определение функции в основном состоит из последовательности инструкций.

Информация о типах во время выполнения программы

Стандарт ANSI C++ описывает два способа поддержки информации о типах во время выполнения программы (runtime type information — RTTI). Первый способ — использование оператора `typeid` (см. также раздел «Оператор `typeid`» в главе «Операторы»). Другой способ — использование оператора `dynamic_cast`, который производит проверку типа объекта во время выполнения программы на предмет совпадения его типа с указателем на требуемый класс (см. также раздел «Проверка типа во время выполнения программы» в разделе «Операторы приведения типов»). В обоих случаях используется информация о типах во время выполнения программы. Некоторые компиляторы, например Microsoft Visual C++, требуют явного указания параметра включения поддержки информации о типах во время выполнения программы.

Исключение

Исключение (exception) — это необычное состояние программы, требующее немедленного вмешательства. Необрабатываемые исключения приводят к аварийному завершению программы. Обычно исключение — это ошибка, происшедшая во время выполнения программы, хотя существуют исключения, не являющиеся ошибками. В ANSI C++ введены ключевые слова `try`, `catch` и `throw` для обеспечения поддержки исключений. Преимуществом обработки исключений в C++ является одинаковый механизм обработки не только ошибок, но и других событий. Таким образом, можно собрать весь код обработки ошибок в одном месте. (См. также разделы «`try`», «`catch`» и «`throw`» в главе «Ключевые слова».)

Исполняемая инструкция

Исполняемая инструкция (executable statement) задает действия программы в процессе ее выполнения (например, какие-либо вычисления или вызов функции). В языке С связь между объявлениями и исполняемыми инструкциями достаточно прозрачна — каждая инструкция есть либо то, либо другое, но не одновременно и то, и другое. В случае С++ ситуация сложнее. Определение переменной, которое создает объект, — не просто определение данных, но и вызов конструктора. Таким образом, в С++ этот термин не так четок, как в языке С. Большинство инструкций, такие как управляющие структуры, вызовы функций, выражения, заканчивающиеся точкой с запятой, могут рассматриваться как исполняемые. Определения данных могут быть, а могут и не быть исполняемыми, но для простоты их можно классифицировать как объявления, а не как исполняемые инструкции.

Класс

Понятие класса (class) является центральным понятием языка С++. Вообще, класс — это определенный программистом тип, к которому относятся любые структуры данных, определенные посредством ключевых слов `class`, `struct` и `union`. При определении класса расширяются возможности языка С++, так как в дальнейшем этот класс можно использовать как любой другой тип С++. Как только класс определен, можно создавать объекты этого класса, каждый из которых имеет члены, определенные в классе (см. также главу «Классы»). Классы основаны на понятии структуры языка С, которая может содержать поля данных, а также расширяться за счет поддержки функций. Эти функции-члены (часто называемые методами) определяют операции над объектами класса. Также можно определить функции операторов (таких как `+`, `*`, `/` и т. д.) для работы с этими объектами (см. главу «Перегрузка операторов»). Другой важной чертой классов является наличие закрытых (`private`) членов, доступ к которым извне невозможен,

более того — невозможно даже узнать об их существовании (см. также раздел «Инкапсуляция»).

Класс памяти

Класс памяти (storage class) переменной или объекта определяет, как компилятор размещает переменную в памяти: в сегменте данных, в стеке или в регистре. Переменные с автоматическим классом памяти размещаются в стеке и не являются постоянными, то есть при выходе из функции такие переменные теряются (уничтожаются). Отметим, что допустимо иметь статический класс памяти для локальной переменной — такая переменная должна определяться как `static`. В языках C/C++ используются следующие классы памяти: `auto`, `extern`, `register` и `static`. (См. также разделы «`auto`», «`extern`», «`register`» и «`static`» в главе «Ключевые слова».)

Ключевое слово

Ключевое слово (keyword) — это имя, определенное в самом языке C++. Сюда не включаются имена, определенные в библиотеках, которые являются идентификаторами. Примеры ключевых слов: `if`, `while`, `do`, `for` и т. д. (См. также описания ключевых слов в главах «Операторы», «Операторы приведения типов» и «Ключевые слова».)

Комментарии

Комментарии (comments) содержат игнорируемый компилятором текст, то есть текст, не попадающий в исполняемый код программы. В комментарии можно привести любой текст — комментарии придуманы для того, чтобы программист мог пояснить свою программу. Язык C++ поддерживает два различных синтаксиса комментариев. Наиболее распространенным является так называемый строковый комментарий, который указывает компилятору проигнорировать весь текст от двойного слэша (`//`) до конца строки:


```
x = y;    // Это – комментарий:  
          // x присваивается значение y
```

Другой вариант, многострочный комментарий, унаследован от языка C. Весь текст между символами `/*` и `*/` игнорируется компилятором (включая символы конца строк). Например:

```
/* Это пример комментария языка C; ,  
являющегося многострочным комментарием. */
```

При использовании многострочных комментариев необходимо помнить, что в них не допускается вложенность. При попытке использовать вложенные комментарии компилятор проигнорирует весь текст от символа начала (`/*`) до первого символа конца комментария (`*/`). Ниже приведен пример, который не будет компилироваться:

```
/* Это внешний комментарий  
/* А это попытка вложенного комментария. */  
Это продолжение внешнего комментария, которое приведет к  
ошибке компиляции. */
```

Если нужно временно убрать блок текста, включающего и комментарии, можно использовать директивы препроцессора `#if` и `#endif`, которые могут быть вложенными. (См. также главу «Директивы, макросы и операторы препроцессора».)

Константа

C++ поддерживает ряд элементов, которые могут быть названы константами (`constant`). В целом константа — это значение, которое не изменяется. Существует несколько различных типов констант.

- Числовые константы, такие как 100, 4.5, 0xFF и 018 — последние две представлены в шестнадцатеричной и восьмеричной системах счисления соответственно.
- Символические константы, определенные директивой препроцессора `#define`, — они также называются макросами.

- Константы, определенные с помощью ключевого слова `enum`.
- Строковые литералы, такие как "Это – строка".
- Имя массива, которое является константой, равной адресу первого элемента (имя массива — константа, но указатели и элементы таковыми не являются).
- Любые переменные, объявленные с ключевым словом `const`.

Последняя категория отличается от всех остальных несколькими существенными моментами. Переменная, объявленная как `const`, остается переменной, несмотря на то, что она константа. Нужно помнить, что переменная с модификатором `const` занимает место в памяти при исполнении программы так же, как и любая переменная. Более того, такая переменная имеет базовый тип, область видимости и класс хранения. Первые три типа констант отличаются от переменных `const`, так как они существуют только в период компиляции программы. В зависимости от конкретного компилятора такие константы могут преобразовываться во время компиляции. Так, выражение `2+2` в большинстве случаев будет заменено на `4`.

Более подробно числовые константы рассмотрены в главе «Основные особенности программирования на C++», строковые константы — в главе «Указатели, массивы и строки», директивы препроцессора — в главе «Директивы, макросы и операторы препроцессора», ключевые слова `enum` и `const` — в главе «Ключевые слова».

Конструктор

Конструктор (`constructor`) — это функция инициализации класса (см. также главу «Конструкторы»). В момент создания объекта вызывается конструктор класса. Конструктор — функция, в которой инициализируются члены класса и выполняются некоторые другие подготовительные дей-

ствия. Конструктор всегда имеет имя, совпадающее с именем класса, и никогда не возвращает никакого значения, даже значение типа `void`. Можно иметь любое число конструкторов для одного класса до тех пор, пока каждый из них имеет свой список параметров (этот список также называется сигнатурой). При инициализации объекта (как при его объявлении, так и при создании с помощью оператора `new`) выбирается и выполняется конструктор с соответствующим списком параметров. При отсутствии параметров вызывается конструктор класса по умолчанию (см. также раздел «Конструктор по умолчанию» в главе «Конструкторы»).

Конструктор базового класса

При создании конструктора класса может понадобиться вызов конструктора базового класса (`base class constructor`). В частности, это особенно полезно в том случае, когда базовый класс имеет закрытые (`private`) члены (без использования конструктора базового класса эти члены не могут быть инициализированы). Общий синтаксис:

```
class::class(args) : base_class(args) { инструкции }
```

В качестве примера рассмотрим класс `CSportsCar` (спортивный автомобиль), производный от базового класса `CAuto` (автомобиль). В следующем примере показан вызов конструктора базового класса (пусть аргументы `h` и `m` передаются конструктору `CAuto`):

```
CSportsCar::CSportsCar(double h, CStr m, double a) :  
    CAuto(h, 2, m)  
{  
    accel_0_60 = a;  
    stripes = 0;  
}
```

В этих строках текста значения `h` и `m` передаются конструктору базового класса `CAuto` вместе с константой 2. (Подобный пример рассмотрен в разделе «Конструкторы базовых классов» главы «Наследование».)

Конструктор копирования

Конструктор копирования (copy constructor) выполняется всегда, когда программа инициализирует новый объект того же класса; другими словами, создается копия этого объекта (см. также раздел «Конструктор копирования» в главе «Конструкторы»). Например, если объект передается некоторой функции по значению, программа должна создать копию объекта и затем эту копию передать в качестве параметра. Другая ситуация возникает при создании копии объекта, который будет возвращен как значение функции (предполагается, что возвращается именно объект, а не указатель и не ссылка на объект). Каждый класс имеет, по крайней мере, один конструктор копирования. Если конструктор копирования для данного класса явно не задан, то компилятор автоматически определит неявный конструктор копирования. Этот конструктор выполнит простое почленное копирование каждого члена данных. Несмотря на то что такое поведение верно для большинства ситуаций, есть ситуации-исключения, в частности при наличии указателей. Конструктор копирования имеет следующий синтаксис, показывающий, что используется ссылка на объект класса *C* (см. раздел «Ссылка»):

```
C(const C&)
```

Конструктор по умолчанию

Конструктор по умолчанию (default constructor) — это конструктор (то есть функция инициализации объекта) без аргументов. При инициализации объекта без параметров используется конструктор по умолчанию. Например, для класса *C* описание конструктора по умолчанию выглядит так:

```
C();
```

Если для класса вообще не написан конструктор, то вызывается конструктор по умолчанию, устанавливающий все переменные в нулевые значения. Однако если имеется хотя бы один конструктор, то компилятор не будет создавать

скрытый конструктор по умолчанию. В связи с этим следует всегда определять конструктор по умолчанию в явном виде. Если говорить более точно, то при обнаружении начала объявления класса компилятор создает конструктор по умолчанию, и, при обнаружении первого же конструктора, он его удаляет. Это необходимо для совместимости со структурами в программах, написанных на С. (См. также раздел «Конструктор по умолчанию» в главе «Конструкторы».)

Косвенность

Термин «косвенный доступ» (indirection) означает обращение к данным по их адресу (указателю) в отличие от прямого доступа. Например:

```
int i = 5;  
int *pi = &i;
```

Указатель `pi` содержит адрес переменной `i`. При использовании выражения `*pi` для доступа к значению `i` выполняется косвенный доступ. Каждое разыменованное есть переход на один уровень косвенности ниже. Например, выражение `**ppi` инициирует переход на второй уровень косвенности (см. также раздел «Разыменованное»).

Литерал

Литерал (literal) — это константа, которая не создает нового символического имени. Литерал обозначает только то, что обозначает, и не более того. Литералы бывают двух типов: числовые (такие как 0, -3, 12, 5.007, 6E20, 0xFF) и строковые, которые представляются текстом, ограниченным двойными кавычками (см. также раздел «Строковый литерал»).

Логические и битовые операторы

В C++ имеется два набора операторов, выполняющих операции математической логики (И, ИЛИ, НЕ). Для одного набора — логических операторов (logic operators) — `&&`

(И), || (ИЛИ), ! (НЕ) — истинным (true) значением является любое ненулевое значение, ложным (false) — нулевое значение. Для другого набора — битовых (или поразрядных) операторов (bitwise operators) — & (И), | (ИЛИ), ~ (НЕ), предназначенных для работы с битами (двоичными разрядами) переменных, объявленных как int или unsigned int, — каждый разряд принимает двоичное значение единица (true) или ноль (false). Не следует путать операторы из этих двух групп (см. также раздел «Булевы значения»). Другое отличие логических и битовых операторов заключается в том, что вычисление логического выражения может выполняться не до конца, если его значение уже определено. Например, если первая половина выражения с оператором И (&&) дала значение ложь (false), то вторая половина не повлияет на результат и ее можно не вычислять. Этот вариант более быстрый, но в нем есть свои «подводные камни»: если во второй части выражения должны вычисляться какие-либо значения, они также не выполняются.

Рассмотрим, например, следующее выражение:

```
(i < j) && (j++ < 10)
```

Если i больше или равно j , выполняется только левая часть выражения, а именно оператор сравнения $i < j$, результатом которого будет ноль (0). Вторая часть этого выражения, $j++ < 10$, в таком случае выполняться не будет, а значит, и значение переменной j не изменится.

Битовые операторы вычисляют выражение всегда до конца.

Макрос

Макрос (macro) — это символическое имя, определенное директивой препроцессора #define. Во время препроцессорной обработки компилятор заменяет каждое определенное с помощью директивы #define имя на соответствующий текст. Например, если PI определено как 3.14159265, то компилятор заменяет каждое вхождение PI на соответствующую числовую константу. Правильное использова-

ние макросов позволяет сделать текст более коротким и удобным для восприятия. Для этого достаточно определить с помощью директивы `#define mytxt` фрагмент текста, занимающий несколько строк. Необходимо отметить, что макросы бывают двух видов: простые, как в определении числа `PI`, и макросы-функции, которые имеют ряд аргументов, подставляемых в соответствующие места при раскрытии макроса (более подробно макросы рассмотрены в главе «Директивы, макросы и операторы препроцессора»).

Массив

Поддержка массивов — одна из важнейших черт практически любого современного языка программирования. Определяя массив (`array`), можно повторить какой-либо тип данных нужное количество раз. После определения массива любой его элемент можно использовать, указывая индекс (позицию) элемента в массиве. Индексация массива предоставляет блестящую возможность его обработки в циклах. Определение массива в виде `type name[n]` создает массив элементов типа `type` от `name[0]` до `name[n-1]`. Например, определение массива `int nums[3]` эквивалентно определению и размещению в памяти компьютера трех переменных типа `int`: `nums[0]`, `nums[1]`, `nums[2]`. В C++, как и в C, существует тесная связь между массивами и указателями. (См. также раздел «Массивы и указатели» в главе «Указатели, массивы и строки».)

Нужно отметить, что имя массива является константой. В предыдущем примере `nums` — это константа, равная адресу элемента `nums[0]`. В отличие от значений элементов, которые могут меняться, адрес массива не изменяется. Следовательно, имя массива также измениться не может. Однако указатель на массив данных может быть изменен.

Метка

Метка (`label`), также называемая инструкцией метки, указывает на точку перехода для инструкции `goto`. Так как

инструкция `goto` редко применяется в программах на C++, то и использование меток — достаточно редкий случай.

Синтаксис:

имя_метки: инструкции

Ключевые слова `case` и `default` также относятся к меткам и используются для обозначения возможных точек перехода, но только в инструкции `switch` (см. также разделы «`switch`», «`case`» и «`default`» в главе «Ключевые слова»). Одной из особенностей C/C++ является то, что инструкция с меткой также является инструкцией. Следовательно, в соответствии с синтаксисом, одна инструкция может иметь несколько меток, например:

```
case 1:
case 2:
case 3:
cout << "Too small\n"
```

Множественное наследование

Согласно стандарту ANSI C++ класс может быть производным от нескольких базовых классов (см. разделы «Базовый класс» и «Наследование»). Рассмотрим пример класса D, производного от классов A, B и C:

```
class D: public A, public B, public C
{
    // определения членов класса D
    ...
}
```

В результате такого определения класса он автоматически получает все члены, определенные в классах A, B и C. Однако множественное наследование (multiple inheritance) может привести к возникновению ряда проблем. Например, если и в классе A, и в классе B есть переменная `x`, то к какой из них выполняется обращение в классе D? Решением ее является использование оператора уточнения — для доступа к члену `x` можно использовать выражение `A::x` (член `x` класса A) или `B::x` (член `x` класса B). В некоторых случаях множественное наследование является очень хо-

рошим решением, но обращаться с ним нужно очень аккуратно.

Надкласс

См. раздел «Базовый класс».

Наследование

Наследование (inheritance) позволяет объявить новый (производный) класс на основе уже существующего (базового) класса, при этом все члены базового класса становятся членами производного. Наследование отражает строгую иерархию классов в ООП (объектно-ориентированном программировании). На основе класса А можно определить класс В, который будет более специализированным, чем класс А. Отношение между ними таково: В является подмножеством класса А. Например, имея класс CAuto, можно определить класс CSportCar, явно указав только необходимые дополнительные члены, а все члены из класса CAuto переносятся автоматически, при этом нет необходимости указывать их заново (см. главу «Наследование»). Наследование является одним из трех столпов объектного программирования, хотя и менее значимым, чем два других. В частности, в компонентной модели объектов (Component Object Model, COM) не придается особого значения наследованию по той причине, что, как черта языка, наследование больше связано с исходным кодом, в то время как COM, в противоположность наследованию, является объектно-ориентированной моделью независимой машины и языка. В действительности наследование весьма полезно: оно эффективно поддерживает повторное использование кода программы. Однако такой же результат можно получить, помещая один объект в другой. В С++ наследование особенно важно из-за того, что является базой виртуальных функций и полиморфизма.

Область действия

Область действия, или область видимости (scope), переменной или функции — это та область, где соответствующий

щий объект определен. В языке C наиболее важны две категории области действия: локальная и глобальная (в программах, текст которых записан в двух и более файлах, возможна также внешняя область действия). Локальная переменная видна только в той функции, где она определена. Вдобавок каждая составная инструкция вводит более узкую область действия: переменная, определенная в составной инструкции, локальна по отношению к этому блоку. Язык C++ добавляет еще одну важную категорию области действия, класс, которая распространяется на данные-члены и функции-члены класса, в котором они определены. Эти члены видны друг другу, функции могут оперировать данными, а при объявлении их как открытыми (public) они могут быть доступны и извне.

Стандарт ANSI допускает ограничение области действия инструкцией `if` путем объявления переменной внутри условия в инструкции `if`. (См. также приложение «Особенности ANSI C++».)

Объект

Объект (object) — это некоторый набор данных, которые могут иметь связанные с ними функции. Объект в этом случае можно рассматривать как «интеллигентную» структуру данных. Каждый объект является реализацией конкретного класса, который определяет поля данных объекта и его поведение. (См. также главу «Классы».)

Объектная ориентация

Объектная ориентация (object orientation) — это метод разработки языков программирования, создания систем и приложений, ориентированный на интегрирование данных и методов работы с ними (в отличие от выделения последних в традиционных языках программирования в отдельные функции или процедуры). Проще говоря, это означает построение программ вокруг объектов, которые имеют как свои свойства (данные), так и методы их обработки (функции). В принципе возможен объектно-ориентированный подход к проектированию даже в традиционных язы-

ках программирования, например, таких как С; однако использование объектно-ориентированных языков типа С++ позволяет существенно облегчить работу, так как в них встроена поддержка классов. Классы С++ определяют в объектах данные-члены и функции-члены (см. раздел «Класс»). В большинстве случаев простейшим способом понимания объектной ориентации является представление программы в виде набора самодостаточных объектов, посылающих сообщения друг другу. Ключевой аспект данной модели заключается в том, что достигается большая степень автономии индивидуальных объектов. Такая модель подчеркивает многоуровневый подход к построению компьютерных систем — каждый уровень «живет своей жизнью», что имеет аналогию в биологии: каждая клетка организма имеет свое внутреннее состояние (данные), но знает, как реагировать на внешние воздействия (методы). Объектно-ориентированное программирование строится на трех основных положениях: инкапсуляции, наследовании и полиморфизме (каждый из этих терминов описан в этом разделе), важнейшим из которых считается инкапсуляция. Инкапсуляция позволяет явно определить взаимоотношения отдельных программных компонентов, то есть объектов, между собой, скрывая подробности реализации. Полиморфизм также важен в определенных системах. Объектная ориентация является естественным шагом в эволюции языков программирования.

Объектно-ориентированное программирование

Объектная ориентация является общей концепцией, а объектно-ориентированное программирование (Object-Oriented Programming, ООР) — это попытка применить ее к конкретному языку или программной архитектуре. Реально же в большинстве случаев оба понятия можно рассматривать как синонимы.

Объявление

Объявление (declaration) — инструкция, которая описывает переменную, класс или функцию, объявляя ее тип.

Некоторые объявления реально создают переменную или тело функции: такие объявления также называются определениями (см. раздел «Определение»), например:

```
int x, y, z; // Определяем x, y и z как целые
```

Другой тип объявления не определяет переменной или функции, но предоставляет информацию, необходимую для правильной интерпретации. Примерами таких определений являются прототипы функций и определения `extern`. В части использования объявлений язык C++ более строг, чем большинство других языков программирования. Например, до вызова функции необходимо ее определить или привести прототип, который полностью описывает возвращаемое значение и типы аргументов. Для понимания сложных объявлений C/C++ может понадобиться обратная логика. Например, чтобы понять объявление

```
char *sarray[100];
```

необходимо знать, что C++ использует в объявлениях те же правила ассоциативности, что и везде. Так как унарные операторы имеют ассоциативность справа налево, то данное объявление эквивалентно объявлению:

```
char *(sarray[100]);
```

В свою очередь, выражение `*(sarray[n])`, в котором `n` может принимать значения от 0 до 99, означает, что это массив из 100 указателей на строки. В отличие от предыдущего объявления

```
char (*ptr_to_array)[100];
```

означает указатель на массив из 100 символов, а не 100 указателей на строки. Используя эту логику, можно понять, чем следующие два объявления отличаются друг от друга. Первое — это прототип функции, а второе — указатель на функцию:

```
// Функция, возвращающая указатель на целое
```

```
int *pfn1(int p1, int p2);
```

```
// Указатель на функцию, возвращающую целое
```

```
int (*pfn2)(int p1, int p2);
```


Для понимания процедуры вызова функции `pfn2` необходимо помнить, что перед ее вызовом, например, с помощью инструкции

```
pfn2 ( a, b); //здесь a и b – переменные типа int
```

указателю `pfn2` должно быть присвоено конкретное имя функции соответствующего типа. Например, такой функцией может быть функция, определенная следующим образом:

```
int sum(int p1, int p2) { return (p1+p2);}
```

Тогда вызов функции `pfn2` выполняется так:

```
pfn2 = sum;  
printf ("Сумма чисел 777 и 125 равна %d", pfn2(777, 125);
```

Операнд

Операнд (operand) — это подвыражение, которое при помощи операторов комбинируется в более сложное выражение. Например, в выражении `x+y` оператор (+) имеет два операнда `x` и `y`. Существуют и выражения с одним операндом, например `*p` (здесь `p` является единственным операндом).

Оператор

Оператор (operator) — это символ, определенный в языке, при помощи которого подвыражения (называемые операндами) комбинируются в более сложные выражения. Большинство, но не все, операторов обозначаются специальными символами, такими как `*`, `+`, `&` и `/`. В некоторых случаях оператор может быть словом (`sizeof`, `new`, `delete`). Некоторые операторы обозначаются двумя символами, например: `++`, `--`, `->` или `==`. Простейший пример использования оператора сложения:

```
x + y
```

Операторы во многом похожи на функции (в главе «Перегрузка операторов» приводится механизм преобразова-

ния операторов в вызов функции). Оператор имеет один или больше входных операндов и одно выходное значение. Но синтаксически они очень отличаются от вызовов функций. Во-первых, в отличие от функций, возможные операторы ограничены лишь теми, которые определены в языке C++. Можно перегружать существующие операторы, придавая им новый смысл, но ввести новые символы операторов нельзя. Во-вторых, операторы имеют свой совершенно отличный от функций синтаксис и не требуют круглых скобок. При их отсутствии последовательность выполнения операторов определяется на основе приоритета и ассоциативности. В главе «Операторы» подробно рассматриваются операторы, их приоритеты и ассоциативность, а в главе «Перегрузка операторов» имеется информация о перегрузке (переопределении) операторов для классов.

Операторы присваивания

Оператор присваивания (assignment operator) позволяет разместить новое значение в переменной (как, например, в выражении $x=1$). Языки C и C++ поддерживают целое семейство операторов присваивания. Большинство из них, такие как $+=$ и $-=$, перед присваиванием значения выполняют еще какую-либо операцию (для указанных операторов это сложение и вычитание). Исключением является оператор, выполняющий только присваивание ($=$).

Другой уникальной чертой языков C и C++ является то, что, согласно синтаксису, присваивание является одним из вариантов выражения, и оно может быть частью другого выражения (это опасно, но допустимо). Как и все выражения, выражение присваивания превращается в инструкцию при добавлении точки с запятой после него:

$x = y;$

Общим свойством всех операторов присваивания является то, что они изменяют содержимое левого операнда. Таким образом, этот операнд должен быть выражением, подобным простой переменной, и иметь для своего значе-

ния определенное место в памяти компьютера. Для таких выражений вводится специальное название `lvalue` (см. раздел «Именуемое выражение»). (См. также раздел «Операции, выражения и комментарии» в главе «Основные особенности программирования на C++» и главу «Операторы», в которой приведен список всех операторов присваивания.)

Определение

Определение (definition) создает переменную или код функции. В большом и сложном проекте функция или переменная может объявляться неоднократно, но определяться только один раз. Существует единственное исключение: виртуальная функция может определяться с одним и тем же именем различным образом в различных классах. Определение переменной — это то место в коде программы, в котором переменная реально создается и, возможно, инициализируется. Например:

```
int i;
```

Определение функции содержит те инструкции, которые должны выполняться при ее вызове. Рассмотрим короткий пример (это пример функции, вычисляющей факториал):

```
long fact(int n)
{
    long amount;
    for (amount = 1; n > 1; n--)
        amount *= n;
    return amount;
}
```

Определение является разновидностью объявления, но не все объявления являются определениями. Так, объявление внешней переменной не определяет переменную.

Параметр

См. раздел «Аргумент (argument)».

Перегрузка операторов

Язык C++ позволяет задать алгоритм работы операторов с классами. Например, можно определить смысл оператора сложения (+) для своего класса, то есть реализовать перегрузку оператора (operator overloading). Эта возможность — одна из самых гибких и интересных в C++. Однако не все объектно-ориентированные языки поддерживают такую возможность, хотя, строго говоря, она не является свойством объектного программирования. (См. также главу «Перегрузка операторов».)

Перегрузка функций

Термин «перегрузка» означает повторное использование символа или имени. Перегрузка функций (functions overloading) — это возможность написать несколько функций с одним и тем же именем. Во избежание конфликта имен все функции должны иметь уникальные списки параметров (фактически имя функции расширяется списком типов параметров, что снимает конфликт). Рассмотрим прототипы трех функций под одним и тем же именем `Init_var`. В каждом случае иницируются различные данные:

```
void Init_var(int);  
void Init_var(double);  
void Init_var(char*);
```

Несмотря на то что все три функции имеют одно и то же имя `Init_var`, каждая является совершенно самостоятельной функцией и должна иметь свое собственное определение. Во время компиляции компилятор анализирует тип параметра, передаваемого в функцию, и выбирает функцию с соответствующим параметром. На перегруженные функции накладываются определенные ограничения. Каждая функция должна иметь уникальную сигнатуру (см. раздел «Сигнатура»), то есть списки параметров должны отличаться по числу и/или типу аргументов. Тип возвращаемого значения функции в понятие сигнатуры не включается, и функции не могут иметь разные возвращаемые

типы при одном и том же списке параметров, но если сигнатуры отличаются, то функции могут иметь различные типы возвращаемого значения. Несмотря на поверхностную схожесть, нельзя путать перегружаемые и виртуальные функции. Основное их отличие в том, что решение о том, какую функцию вызвать при перегрузке функции, принимает компилятор во время компиляции программы, а решение о переопределении виртуальной функции принимается уже во время работы программы.

Переменная

Переменная (variable) — это именованный фрагмент памяти для сохранения данных. Если при описании переменной не использовалось ключевое слово `const`, то в переменной можно сохранять различные данные (в противном случае невозможно изменить занесенное при инициализации значение). Для обозначения переменной допускаются практически любые имена, не совпадающие с ключевыми словами (см. раздел «Идентификатор»). В C++ переменные могут сохранять простейшие данные, указатели или объекты классов. Как в C, так и в C++ все переменные должны быть определены до их использования.

Перечисление

Перечисление (enumeration) — это синоним слова «список». В программах на C++ перечисление — это быстрый и легкий способ создать список целых констант. Можно написать целый ряд директив `#define` для определения этих констант, однако при помощи ключевого слова `enum` этого можно добиться быстрее и элегантнее (см. также раздел «enum» в главе «Ключевые слова»). Вот простой пример (здесь названия совпадают с их числовыми значениями, так как перечисление начинается с нуля):

```
enum number
{
    zero, one, two, three;
};
```


Более жизненный пример: припишем номера от 0 до 3 карточным мастям. Тогда ваша программа станет более понятной:

```
enum suit  
{  
    CLUBS, DIAMONDS, HEARTS, SPADES;  
};  
// ...  
cout << HEARTS << endl; // напечатает "2"
```

Подкласс

См. раздел «Производный класс».

Полиморфизм

Одной из основополагающих идей объектно-ориентированного программирования является максимально возможная независимость каждого объекта (инкапсуляция). Тогда внутренняя структура объекта, а значит, и детали определения его функций могут быть скрыты для всех остальных объектов. Как в таком случае организовать взаимодействие этих объектов? Одним из способов является обмен сообщениями (messages). Структура этих сообщений не зависит от конкретных объектов, которым или от которых они передаются. (Как правило, сообщения посылаются при возникновении каких-либо событий, например нажатии кнопки мыши, срабатывании таймера, нажатии клавиши на клавиатуре.) Имя функции, обрабатывающей одно и то же событие, для всех объектов одинаково, но ее реализация (определение) зависит от объекта, точнее, класса этого объекта и определяется во время выполнения программы.

Рассмотрим еще одну ситуацию. Допустим, на экране есть некоторый набор команд, каждая из которых выполняет некоторые действия после того, как пользователь выбрал эту команду и инициализировал ее выполнение (обычно это делается нажатием клавиши Enter или щелчком на кнопке мыши). В этой ситуации удобно определить каждую команду в виде объекта, а функцию-член этого объекта,

реализующую действия команды, назвать одним именем для всех объектов. Конкретная реализация функции (с учетом объекта) будет определяться во время выполнения программы (см. также раздел «Определение виртуальной функции» в главе «Виртуальные функции»).

Это свойство называется полиморфизмом (polymorphism), что в переводе с греческого означает «много форм». C++ поддерживает полиморфизм посредством виртуальных функций, которые могут имитироваться в языке C при помощи указателей на функции. Более подробное объяснение механизма работы виртуальных функций приведено в главе «Виртуальные функции».

Многие современные операционные системы являются полиморфными, так как для управления программами (иногда их называют задачами или приложениями) они используют сообщения, ничего (или почти ничего) не зная о самих задачах, которым посылаются эти сообщения. Более того, реально задача может быть запущена на выполнение уже после отправки сообщения. Примером такой полиморфной системы может служить операционная система Microsoft Windows.

Полиморфизм является одним из трех столпов объектно-ориентированного программирования — наряду с инкапсуляцией и наследованием.

Прагма

Под термином «прагма» (pragma) понимается команда препроцессора, устанавливающая некоторое условие для компилятора и целиком зависящая от конкретной реализации этого компилятора. По своей природе прагмы приводят к созданию непереносимых программ, хотя обойтись без них иногда сложно. (См. также раздел «Директивы, макросы и операторы препроцессора».)

Приведение типа

Приведение типа (cast) заключается в том, что берется значение выражения, преобразуется к требуемому типу и

передается для дальнейшей обработки. Если приведение типа производится для переменной, то значение переменной остается без изменения, однако в некоторых случаях при этом изменяется внутреннее представление значения переменной в памяти компьютера. Например, приведение типа целого значения к вещественному изменяет его внутреннее представление:

```
double d = static_cast <double> (i);
```

В других случаях приведение типа не влияет на внутреннее представление, а только изменяет способ интерпретации выражения. Так, приведение типа `int*` к типу `float*` не воздействует ни на внутреннее представление значения, ни на представление указателя, но в дальнейшем при использовании значения можно получить печальные последствия (см. также главу «Операторы приведения типов»). В ANSI C++ поддерживается ряд новых операторов приведения типов, таких как `static_cast`, но также поддерживается и старый стиль.

Приоритет

Приоритет (precedence) определяет порядок выполнения операторов в сложном выражении. Операторы с большим приоритетом выполняются раньше. Например, при вычислении следующего выражения выполняется три шага:

```
p = x + 2 * n
```

Порядок выполнения: сначала умножение (`*`), затем сложение (`+`) и присваивание (`=`) (оператор присваивания является таким же равноправным оператором, как и остальные). Согласно этому правилу выражение вычисляется следующим образом:

```
p = (x + ( 2 * n))
```

Как и в большинстве других языков программирования, в C++ используется «интуитивно» понятный порядок приоритетов, соответствующий порядку выполнения действий в математике. Так, среди математических операторов (`+`, `-`, `*`, `/`) используется стандартное соглашение о том, что

операторы умножения и деления имеют больший приоритет, чем операторы сложения и вычитания. Следующий уровень приоритета занимают операторы отношения и логические операторы. Операторы присваивания имеют самый низкий приоритет, в противоположность унарным операторам, которые имеют наивысший приоритет. Если два соседних оператора имеют одинаковый приоритет, то C++ использует правило ассоциативности для разрешения этого конфликта. В табл. 3 и 4 (см. главу «Операторы») представлен список приоритетов всех операторов.

Продвижение

Продвижение (promotion) — это то, что «грозит» программисту, который успешно пишет 1000 строк программного кода, не содержащего ошибок, в день или (еще более страшно) может успешно объяснить своему шефу смысл термина «полиморфизм».

Продвижение означает процесс автоматического преобразования «меньших» типов данных к «большим» типам. Такой механизм используется при обработке выражений, содержащих различные типы операндов. Например в случае присваивания целого значения переменной с плавающей точкой, как в случае `float f = 1`. В этом случае C++ автоматически преобразует целое значение в вещественное. Другим примером является приведение обоих операндов к типу `long`, если хотя бы один из них имеет тип `long`. В таких случаях не нужно явного приведения типов. В C и C++ существует иерархия продвижения типов. Например, `short` продвигается при необходимости к `long`, а `long` — к `double`.

Производный класс

Производный класс (derived class) наследует некоторые или все свои члены от другого класса, называемого базовым классом. Например, если класс C является производным от базового класса B, то все члены класса B автоматически становятся членами класса C. При этом класс C может

содержать и свои собственные члены. Иногда производный класс называют подклассом, или дочерним классом. (См. также раздел «Создание производного класса» в главе «Наследование».)

Простейшие типы данных

Простейшие типы данных (primitive data type), иногда называемые примитивными типами, — это набор типов данных, определенных в самом языке C++. Сюда включаются целочисленные типы (такие как `char`, `int`, `unsigned int` и `long`), типы чисел с плавающей точкой (`float` и `double`) и определенный в стандарте ANSI тип `bool`, который хотя и не используется как целое, но содержит два целых значения: 0 и 1. Все простейшие типы C++ являются числовыми. Строка не является простейшим типом данных в C++, так как она представляется массивом символов типа `char`.

Простейшие типы очень важны в C++, так как все другие типы (включая классы и массивы) формируются исключительно из простейших типов. Они являются фундаментальными строительными кирпичами при построении структур данных. Такие структуры, как классы и массивы, иногда называются сложными (или составными) типами.

Пространство имен

Пространство имен (namespace) является фундаментальной концепцией C++. Пространство имен — это группа имен, в которой имена не совпадают. Исключением являются имена перегружаемых функций и переменные с различными областями действия (см. разделы «Перегрузка функций» и «Область действия»). Имена в различных пространствах имен не конфликтуют. Например, можно использовать имя `my_address` в двух различных классах (это же относится к структурам и объединениям). C++ также позволяет определить пространства имен при помощи ключевого слова `namespace`. Такие пространства имен вводятся для снижения вероятности конфликта имен и полезны в

случае использования имен из нескольких различных библиотек. (См. раздел «namespace» в главе «Ключевые слова».)



ПРИМЕЧАНИЕ Имена также называют идентификаторами.

Прототип

Прототип (prototype) — определение функции, содержащее полную информацию о количестве и типах параметров, а также о типе возвращаемого значения. В C++ в отличие от C каждое определение функции должно иметь полный прототип.

Прототипы можно использовать для определения функций, которые будут описаны ниже в данном тексте или даже в другом тексте. В таких прототипах не обязательно указывать имена формальных параметров, хотя их использование и допустимо. Вот пример прототипа функции, принимающей строковый аргумент и возвращающей целое значение:

```
int print_items(char*);
```

Слово «прототип» имеет отличный от обычного смысл среди системных аналитиков и специалистов по пользовательскому интерфейсу. Для них прототип системы означает только создание пользовательского интерфейса для демонстрационных целей, а следовательно, ничего не делающего. Такое понимание не имеет ничего общего со смыслом, который вкладывается в него в C++. (Отличия прототипа от заголовка определения функции см. в разделе «Функции» главы «Основные особенности программирования на C++».)

Прототип функции

См. раздел «Прототип».

Пустая инструкция

В языках C и C++ допустимо использовать пустую инструкцию (empty statement), которая состоит из одного при-

Реализация

Термин «реализация» (implementation) имеет несколько смыслов:

- реализация может означать определение функции. Если создается производный класс и определяется перегружаемая или виртуальная функция, то это называется реализацией функции;
- реализация может относиться к внутренним членам классов, в противоположность открытым (public) членам;
- термин специфика реализации может относиться к специфике реализации конкретного компилятора C++.

Заметим, что в C и C++ точка с запятой (;) является ограничителем инструкции, но не разделителем составных инструкций, как в языке программирования Паскаль (Pascal). Единственное, что не нужно заканчивать точкой с запятой, — это саму составную инструкцию (хотя хуже от этого не будет). Единственным исключением является определение класса (см. раздел «Создание нового класса» в главе «Классы»).

Рекурсия

Рекурсивная (recursion) функция — это функция, вызывающая саму себя. Конечно, можно определить функцию так, что рекурсия будет бесконечной (а потом либо компьютер «зависнет», либо задача будет удалена как ненужная), но реально необходим какой-либо механизм для обнаружения конца рекурсии. Классическим примером рекурсивной функции является функция вычисления факториала:

```
long fact ( int n)
{
    if ( n > 1)
        return (n*fact ( n-1));
    else
        return 1;
}
```


С++ полностью поддерживает рекурсию, но необходимо помнить об особенностях применения локальных статических переменных, которые могут привести к совершенно неожиданным эффектам.

Сигнатура

Сигнатура (signature) однозначно идентифицирует символ по его имени, типу и, в случае функции, его списку аргументов. Необходимо отметить, что имена формальных параметров в данном случае не имеют значения. Функции `init(int i)` и `init(int j)` имеют одинаковые сигнатуры с точки зрения компилятора. С другой стороны, функции `init(int)` и `init(double)` различны. При генерации объектных или исполняемых файлов компилятор использует механизм декорирования для формирования имен вместе с их сигнатурой.

Символ новой строки

Символ новой строки (newline) — это специальный символ (изображаемый в строковых литералах и символьных константах как `\n`), который обозначает конец строки. Вывод этого символа на дисплей приводит к перемещению курсора (и, соответственно, выводимого далее текста) на первую позицию новой строки. При работе с потоковыми объектами ввода/вывода (такими как `cin` и `cout`) можно заменить этот символ на оператор `endl`. (См. также примеры в главе «Объекты и классы библиотеки ввода/вывода».)

Сложные типы данных

Сложный тип данных (complex data type) — это тип данных, построенный из других типов. Сложные типы данных включают в себя все массивы, классы, объединения и структуры. Только простейшие (примитивные) типы данных (`int`, `bool`, `float` и подобные) не относятся к сложным типам данных.

Создание экземпляров класса

Создание экземпляров класса (instantiation) — то же, что создание объектов этого класса. Вообще, класс можно рассматривать как понятие, введенное специально для определения объектов, однако не для каждого класса можно создать его объекты. В частности, невозможно создание объектов абстрактного класса, по крайней мере, напрямую.

Составные инструкции

Составная инструкция (compound statement) состоит из одной или нескольких инструкций, ограниченных фигурными скобками (`{ ... }`). Такие инструкции работают как единое целое, то есть исполняются либо все, либо не исполняется ни одной.

Синтаксис:

```
{ инструкции }
```

Составные инструкции применяются в различных целях. Во-первых, составная инструкция определяет блок инструкций. Локальные переменные, определенные внутри блока, не видны вне его, то есть блоки можно использовать для управления видимостью переменных. Во-вторых, составные инструкции используются для передачи управления группе инструкций в управляющих структурах. Составную инструкцию можно написать везде, где можно написать обычную инструкцию.

Ссылка

Переменная ссылки (reference) — это псевдоним для другой переменной. Например:

```
int b = 1;  
int &a = b; // a – псевдоним для b  
a = 5; // b получает значение 5
```

Аргумент ссылки является псевдонимом для переменной, которая передается в функцию. Например:


```
void switch_values(int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

В этой функции не используются указатели. Тем не менее в такой версии вызов можно записать без использования операторов над указателями (& и *):

```
int x = 1, y = 2;
switch_values(x,y); // После вызова x = 2, y = 1
```

При использовании ссылок необходимо помнить, что реально передаются указатели, хотя из синтаксиса этого не видно (что подобно передаче по ссылке в BASIC и FORTRAN). Аналогично, возвращаемое значение ссылки означает, что функция возвращает указатель на объект, но не сам объект. Ссылки оказываются полезными в синтаксисе функций-членов, таких как конструкторы копирования и функции операторов присваивания. Это послужило причиной включения ссылок в язык C++. (См. также главы «Конструкторы» и «Перегрузка операторов».)

Ссылка вперед

В языке C++ все его элементы (переменные, имена или классы) должны быть определены до момента их использования. Однако, если нужно вызвать функцию до ее определения, можно создать прототип функции в начале программы или непосредственно перед ее использованием. Аналогичным образом C++ позволяет указать класс до его полного определения. Такая необходимость может появиться в тех редких случаях, когда два класса содержат ссылки друг на друга. Такие ссылки допустимы, но, по крайней мере, для одного класса должно быть сделано предварительное определение или ссылка вперед (forward reference). Для этого достаточно просто написать

```
class <имя класса>;
```


в начале программы. Для структуры нужно написать

```
struct <имя структуры>;
```

Например:

```
class A;  
class B;  
//...  
class A {public: B *ptr_to_B;};  
class B {public: A *ptr_to_A;};
```

ANSI C++ расширяет возможность предварительного определения на классы, вложенные в другие классы (см. разделы «Управляющая структура» и «Класс»).

Строка

Строка (string), также называемая текстовой строкой или строкой символов, представляет собой массив символов.

Более подробно строки рассматриваются в главе «Указатели, массивы и строки», библиотечные функции работы со строками рассматриваются в разделе «Функции работы со строками» главы «Библиотечные функции», а кодирование специальных символов рассмотрено в главе «Типы данных».

Строковый литерал

Строковый литерал (string literal) состоит из текста, заключенного в двойные кавычки ("). Строковый литерал является одной из главных категорий констант в C++ (см. раздел «char» в главе «Типы данных»). Пример строкового литерала:

```
"Hello, C++!"
```

Компилятор C++ обрабатывает строковые литералы особым способом. Во-первых, он резервирует место в сегменте константных данных и помещает туда текст. Затем он заменяет литерал в программе на адрес размещенной строки (но сам исходный текст программы, естественно, не изменяется!). Такое поведение компилятора позволяет рас-

смаатривать все строки как массивы символов и использовать вместо строк адреса их первых элементов (символов).

Текстовая строка

См. раздел «Строка».

Указатель

В языках С и С++ одним из центральных является понятие указателя (pointer). Указатель — это переменная, которая содержит адрес.

В большинстве языков программирования, таких как BASIC, FORTRAN или PASCAL, программы очень интенсивно работают с адресами — просто это скрыто от пользователей (PASCAL имеет переменные типа указателей, но их возможности сильно ограничены). Предоставляя программисту возможность работать с адресами памяти, С и С++ требуют от программиста хорошего понимания того, что он делает, что делает программа, а также внимания и аккуратности. С другой стороны, работа с адресами предоставляет широкие возможности для оптимизации программы. Наиболее очевидным примером оптимизации является передача адреса данных, а не самих данных (что особенно важно при работе с большими объемами данных). При этом имеется возможность модифицировать эти данные. Благодаря своим особенностям указатели в программах на С/С++ имеют очень широкое применение.

Указатель на функцию

Одним из удачных решений в языках С и С++ является определение указателя, содержащего адрес функции (в отличие от адреса данных). Указатели на функцию (pointer to a function) имеют достаточно необычные определения, подобные представленному ниже. Здесь показан пример указателя, имеющего имя `pf` и указывающего на функцию, принимающую два целочисленных аргумента и возвращающую значение типа `double`:


```
double (*pf)(int,int);
```

Указатели на функции нужны для описания функций обратного вызова. Так, для вызова библиотечной функции `qsort` ей необходимо передать адрес функции, которая производит сравнение двух элементов, передаваемых из функции `qsort` (см. главу «Библиотечные функции»). Другая ситуация, когда возникает необходимость в указателях на функцию, — это задача построения массивов адресов функций, что, в свою очередь, предоставляет возможность индексированного вызова функций. Таким образом, в языке С можно получить эффект виртуальных функций.

В общем, именно указатели на функции и адреса позволяют эмулировать механизм виртуальных функций в С (используя, правда, другой синтаксис). Это возможно благодаря тому, что указатель на функцию можно определять уже во время выполнения программы, а не на этапе компиляции.

Управляющая последовательность

Управляющая последовательность (escape sequence) — это способ кодирования специальных (например, управляющих выводом) символов в строках или символьных константах. Эти последовательности (такие как `\n`, которая означает переход к новой строке) начинаются с символа обратного слэша (`\`). Для обозначения самого обратного слэша используется двойной обратный слэш (`\\`). (Список управляющих последовательностей приведен в табл. 2 главы «Типы данных».)

Управляющая структура

Управляющая структура (control structure) — это инструкция, имеющая возможность изменять ход выполнения программы на основе каких-либо условий. Управляющими структурами в С++ являются инструкции `if-else`, `while`, `do`, `for` и `switch`. Сюда также можно отнести функции и инструкцию `goto`. (См. также раздел «Функции» в теку-

щей главе и соответствующие разделы в главе «Ключевые слова».)

Условная компиляция

Условная компиляция (conditional compilation) — это способ поддержки нескольких версий одной программы. Чтобы не переписывать программу для каждой новой платформы, можно поместить строки текста, зависящие от платформы, в блоки `#if...#endif`. (См. раздел «`#if`» в главе «Директивы, макросы и операторы препроцессора».)

Файлы заголовков

Файлы заголовков (header files) обеспечивают общие определения, а также прототипы функций. Язык C++ требует, чтобы каждая функция имела явный прототип до ее использования. Более того, многие функции стандартной библиотеки реализованы в виде макросов. Вся эта информация должна быть прочитана в компилируемый файл до ее использования. Файлы заголовков, в которых находится требуемая информация, позволяют избежать непосредственного внесения этой информации в исходные тексты программы, что привело бы к значительному увеличению текстов программ. Для включения файла заголовков в файл текста программы требуется директива `#include` (см. также главу «Директивы, макросы и операторы препроцессора»). В ANSI C++ появилась новая, более предпочтительная, техника использования не библиотечных файлов заголовков, а виртуальных файлов заголовков. Например, вместо директивы `#include <stdio.h>`; лучше указать `#include <cstdio>`;. Это позволит в последующих версиях C++ отказаться от физических файлов заголовков типа `stdio.h` и использовать встроенные объявления, существенно ускоряющие процесс компиляции программы. Файлы заголовков полезны для проектов, содержащих несколько файлов исходных текстов. В файлы заголовков рекомендуется помещать следующую информацию:

- прототипы всех функций, кроме тех, которые используются только в одном модуле;

- описания всех классов проекта;
- описания всех внешних (extern) переменных, используемых для обмена информацией между модулями;
- определения макросов;
- определения сложных типов, используемых во всем проекте.

При разработке программы, для которой требуется несколько файлов заголовков, из них необходимо исключить повторяющуюся информацию, так как это может привести к ошибкам при компиляции и в процессе выполнения программы. Например, определение функции, данное в двух или более файлах, приведет к ошибкам при построении исполняемого файла.

Функции

Функция (function) — группа инструкций, исполняемая как единое целое и предназначенная для выполнения некоторых действий или получения значения. Концепция функции является центральной во всех языках программирования, хотя в других языках функции могут называться процедурами или подпрограммами. В C/C++ любые подпрограммы называются функциями, независимо от того, возвращают они значения или нет. Функции, не возвращающие значения, объявляются как `void`. Но реально разница между функциями, возвращающими и не возвращающими значение, отсутствует. Если функция возвращает значение, но оно не нужно, его можно не использовать. (См. также раздел «Функции» в главе «Основные особенности программирования на C++».)

Функции обратного вызова

При вызове некоторых функций им необходимо передать адрес какой-либо другой функции, которая уже определена. Последние называются функциями обратного вызова (или callback-функциями — `callback function`). Обычно функциям обратного вызова временно передается управ-

ление для выполнения некоторых действий, а затем они возвращают управление в вызвавшую их функцию. Такое решение является единственным путем использования функций быстрой сортировки (`qsort`) и двоичного поиска (`bsearch`) с достаточной степенью гибкости (см. также разделы «`qsort`» и «`bsearch`» в главе «Библиотечные функции»). При вызове такой функции ей передается адрес вызвавшей функции.

Функции преобразования

Язык C++ позволяет выполнять преобразование типов двумя способами. Во-первых, класс `C` может быть построен из типа `T` с помощью конструктора `C(T)`. Экземпляр класса `T` может быть присвоен экземпляру класса `C` (в инструкции `c_obj = t_obj`). Во-вторых, функции преобразования (`conversion functions`) определяют преобразования класса к другому типу. Например, если класс `C` имеет функцию преобразования к типу `T`, то эта функция определяет, как экземпляр класса `C` может быть присвоен экземпляру класса `T` (в инструкции `t_obj = c_obj`). (См. также раздел «Создание функции преобразования» в главе «Перегрузка операторов».) Конструкторы и функции преобразования определяют больше, чем просто действия при присваивании; они также обеспечивают явное преобразование типов. Например, если класс `MyStr` определяет преобразование к типу `char*`, то можно передавать экземпляр класса `MyStr` (в качестве параметра, например) в любом месте, где тип параметра функции определен как `char*`.

Функция-член

Функция-член класса (`member function`) — это функция, определенная внутри класса (во многих книгах также используется ее синоним — метод). Функции-члены определяют поведение объектов этого класса (исключением являются статические функции-члены, которые применяются именно к классу, а не к его отдельным объектам). Функции-члены имеют своей областью действия сам класс или его объекты. Функция-член может быть вызвана без ука-

зания своего класса или объекта только внутри функций-членов этого класса, во всех остальных случаях она вызывается только с указанием объекта, для которого она выполняется. Функция-член имеет доступ и к собственным данным-членам этого класса. Рассмотрим пример:

```
CPnt aPoint;  
aPoint.move(10, 20);
```

Здесь функция `move` воздействует на данные объекта `aPoint` класса `CPnt`. (См. также главу «Классы».)

Чистая виртуальная функция

Как ни странно это прозвучит, но чистая виртуальная функция (*pure virtual function*) — это функция, не содержащая кода. Чистая виртуальная функция действует как заготовка или общий прототип для кода, определяемого в других классах. Любой класс, который содержит чистую виртуальную функцию, является абстрактным классом, и, соответственно, создание объекта такого класса невозможно. При использовании таких классов как базовых создаются производные классы, в которых реализуются какие-либо или все чистые виртуальные функции.

Чистые виртуальные функции используют запись `=0`. В следующем примере определяется чистая виртуальная функция `report_stats`:

```
virtual void report_stats(int, char*) = 0;
```

Определение чистой виртуальной функции содержит полную информацию о типах.

Член

Член (*member*) — это переменная или функция, определенная внутри класса (см. также разделы «Функция-член» и «Данные-члены»).

Шаблон

Шаблон (*template*) — объявление, содержащее родовой тип. Этот тип, который можно обозначить идентификато-

ром (например, `T`), является заготовкой для типа, определяемого позднее. Будучи объявленным, шаблон может использоваться для генерации специального объявления, замещающего тип `T` реальным типом. Например, следующее объявление шаблона позволяет создать структуру данных под названием `pair`, в которую входят два объекта типа `T`:

```
template <class T>
class pair
{
    T a, b;
}
```

Как только это объявление прочитано компилятором, шаблон `pair` может использоваться для объявления специальных типов, например `pair<int>`:

```
pair<int> jeans;
```

Теперь структура `jeans` является классом, содержащим два целых `a` и `b`. (См. также раздел «`template`» в главе «Ключевые слова».)

Экземпляр

Экземпляр (`instance`) — синоним понятия «объект». Как только класс объявлен, можно использовать его для создания любого количества экземпляров. В некотором смысле отношения между классом и его экземпляром напоминают отношения между типом `int` и конкретным значением целого типа. Класс определяет тип и определения его функций-членов, в экземпляре класса находятся значения данных-членов.

Экземпляр класса

Экземпляр класса (`class instance`) — это то же, что объект класса. (См. также разделы «Экземпляр» и «Объект».)

Типы данных

Язык C++ поддерживает несколько базовых типов данных, которые также называют простыми (иногда арифметическими) типами данных. Эти типы определены в самом языке, и все остальные типы строятся на их основе. В табл. 1 приводятся все простейшие типы данных, определенные в C++.

Следует отметить, что в зависимости от версии компилятора область изменения чисел может меняться. Точные пределы изменения чисел приводятся в файлах заголовков `limits.h` и `float.h`.

Таблица 1. Простейшие типы данных C++

Тип	Описание	Область изменения
bool	Булево (логическое) значение	true или false
char	1-байтовое целое, используемое для хранения символа	От -128 до 127 или от 0 до 255
unsigned char	1-байтовое беззнаковое целое	От 0 до 255
signed char	1-байтовое целое	От -128 до 127
int	Целое (2 или 4 байта)	short или long
unsigned int	Беззнаковое целое	От 0 до 65 535
short	2-байтовое целое	От -32 768 до 32 767
unsigned short	2-байтовое беззнаковое целое	От 0 до 65 535

Тип	Описание	Область изменения
long	4-байтовое целое	От -2 147 483 648 до 2 147 483 647
unsigned long	4-байтовое беззнаковое целое	От 0 до 4 294 967 295
float	Действительное число (4 байта)	От $-3.4 \cdot 10^{+38}$ до $-3.4 \cdot 10^{-38}$ и от $3.4 \cdot 10^{-38}$ до $3.4 \cdot 10^{+38}$; точность — 7 десятичных цифр
double	Действительное число с двойной точностью (8 байт)	От $-1.7 \cdot 10^{+308}$ до $-1.7 \cdot 10^{-308}$ и от $1.7 \cdot 10^{-308}$ до $1.7 \cdot 10^{+308}$; точность — 15 десятичных цифр
long double	Длинное действительное число с двойной точностью (10 байт)	от $-3.4 \cdot 10^{+4932}$ до $-3.4 \cdot 10^{-4932}$ и от $3.4 \cdot 10^{-4932}$ до $3.4 \cdot 10^{+4932}$; точность — 19 десятичных цифр
wchar_t	2-байтовый символ, используемый в международной кодировке, такой как Unicode	От 0 до 65 535

Следующие замечания поясняют некоторые из особенностей использования типов в C++.

- Строго говоря, signed, unsigned, short и long являются не самостоятельными типами, а модификаторами. Тип int определяется в C++ по умолчанию, таким образом, unsigned int и unsigned — это один и тот же тип, и оба варианта записи являются допустимыми в C++. Аналогичным образом можно записать signed или signed int, short или short int, long или long int.
- По умолчанию все целые типы C++ определяются как signed, за исключением типа char. Таким образом, записи signed int и int определяют один и тот же тип.

В спецификации ANSI область изменения целых чисел определена от -127 до 127 . Однако использование дополнительного кода (см. также раздел «Дополнительный формат» в главе «Словарь терминов и понятий») увеличивает эту область на единицу, и минимальное целое число равно -128 . Числа в плавающем формате не предоставляют возможности точного задания числа 0. Точность определения нуля зависит от выбранного формата числа. Например, число $1E-50$ (единица, деленная на 10 в степени 50) не может быть задано типом `float`, но может быть определено типом `double`.

Все базовые типы C++, включая символьный и булев, являются числовыми, так как все они, несмотря на различные форматы, в памяти компьютера представлены двоичными числами.

За исключением `float`, `double` и `long double`, все базовые типы являются целыми, что означает отсутствие у них дробной части. Большинство типов, при отсутствии модификатора `unsigned`, могут определять как положительные, так и отрицательные числа (см. также раздел «Дополнительный формат» в главе «Словарь терминов и понятий»).

Константы любого целого типа могут быть заданы в шестнадцатеричной или восьмеричной системе счисления. Для определения шестнадцатеричной константы используется префикс `0x` или `0X`. Запись такой константы может содержать цифры от 0 до 9, а также символы шестнадцатеричных цифр A, B, C, D, E, F или a, b, c, d, e, f. Признаком восьмеричной константы является начальный 0. В ее записи допустимы только цифры от 0 до 7. В остальных случаях константа считается представленной в десятичной системе счисления.

Следующие три оператора присваивания задают одно и то же значение:

```
A=0xFFFF; A=077777; A=4095;
```

Если в записи константы встречается суффикс `L` или `l`, компилятор интерпретирует константу как `long`, то есть

размещает ее в 4-байтовом слове. Суффикс U (u) указывает на константу типа unsigned. Разрешается комбинировать оба эти суффикса в любом порядке для задания константы типа unsigned long.



ВНИМАНИЕ Константы типа int и long имеют разное представление в памяти компьютера: первые занимают два байта, вторые — четыре байта. Будьте особенно внимательны при задании функций, имеющих параметры типа long, не забывайте указывать суффикс L при их записи.

Язык C++ поддерживает два варианта записи констант с плавающей точкой (иногда говорят — с плавающей запятой): с десятичной точкой (.) в качестве разделителя и экспоненциальную форму, в которой используются символы e или E (для указания числа 10 в качестве основания степени), за которым следует показатель степени.

По умолчанию константы с плавающей точкой имеют тип double. При наличии суффикса F (f) константа имеет тип float, L (l) — тип long double.

Следует отметить два преимущества работы с целыми числами:

- операции с ними выполняются быстрее;
- они хранят точные целые значения, что позволяет избежать возможных ошибок при округлении.

Тем не менее типы чисел с плавающей точкой являются более гибкими, и во многих случаях их использование необходимо. Следующий пример демонстрирует получение неправильного результата при неверном выборе типов переменных:

```
int a=5, b=2, c;
c = a/b*b; //c равно 4, а не 5
```

Несмотря на то что мы сначала делим переменную a на b, а затем умножаем результат деления на то же число b, результат не будет равен делимому, так как при выполнении арифметических операций над целыми числами резуль-

татом считается целое число. Ошибки можно было бы избежать, определив тип хотя бы одного из операндов как `float`. Рассмотрим теперь каждый из типов более подробно.

bool

Тип данных `bool` (булев) определяет ровно два значения `true` и `false`. Переменные этого типа в основном используются в двух случаях: при определении сложных условий в условных инструкциях и в качестве флагов в инструкциях цикла:

```
//использование булевой переменной
//в условной инструкции
bool bw1, bw2;
//...
bw1 = (x2-x1)/scalex > 100;
bw2 = x2 < 600;
if ( bw1 && bw2 ) myf1 ();
//использование булевой переменной
//в инструкции цикла
bw1=true;
while (bw1)
{
    //...
    if (x2 < 600) bw1 = false;
    //...
}
```

Обычно в С и С++ значения `true` и `false` присваивались переменным целого типа. Сейчас более предпочтительным вариантом является использование булевого типа данных для этих значений. Булевой переменной можно присвоить любое целое значение (компилятор выдаст предупреждение, если при этом не выполнено приведение типов), но любое ненулевое значение при этом автоматически преобразуется в `true` (1).

Любое целое ненулевое значение рассматривается как `true` в условных инструкциях и инструкциях цикла. Однако результатом операции сравнения или логической операции может быть только 0 (ноль) или 1 (единица).



ПРИМЕЧАНИЕ В С и ранних версиях С++ булев тип не поддерживался.

char

В памяти компьютера тип данных `char` представляется, как и другие целые типы, числом, однако в действительности это число является кодом символа. Преобразование символа в число и обратно происходит автоматически в соответствии с таблицей кодировки символов ASCII, представленной в приложении «Таблица ASCII-кодов». Например, код символа «равно» (=) представляется в памяти компьютера как число 61 и занимает один байт. При его обозначении в качестве простого символа он должен быть заключен в апострофы.

На практике тип `char` довольно редко используется для хранения отдельных символов и значительно чаще для хранения строк, являющихся массивами символов. Инициализировать массив символов можно строкой, например, показанная ниже строка отводит в памяти компьютера 20 байт и заполняет 6 из них символами, заданными в строке:

```
char cstr[20]="строка";
```

Следующий, 7-й, символ отводится для признака конца строки (`\0`), который называется нуль-терминатором. Опустим в этом примере длину массива и запишем инструкцию следующим образом:

```
char cstr[]="строка";
```

В этом случае в памяти компьютера будет отведено ровно 7 байт, включая нуль-терминатор, и все они будут инициализированы.

В отличие от одного символа строка заключается в кавычки и занимает в памяти компьютера число байт, равное количеству символов в ней, плюс один для признака конца строки.

Тип `char` эквивалентен типу `signed char` или `unsigned char` в зависимости от соответствующего параметра компилятора.



ВНИМАНИЕ При работе со строками переменной длины следует внимательно следить за их длиной, так как автоматического контроля за переполнением массива, отведенного для строки, не производится. Таким образом, копирование строки в слишком короткий массив, отведенный для нее, может привести к порче данных в памяти.



ПРИМЕЧАНИЕ Некоторые кодировки символов, например Unicode, требуют для каждого символа два байта. Требованиям таких кодовых таблиц удовлетворяет тип `wchar_t`.

Для управления выводом строки в ней могут быть использованы управляющие последовательности (управляющие символы), представленные в табл. 2. Наиболее часто используемым среди них является символ новой строки (`\n`). Обратный слэш (`\`) в C++ всегда является признаком управляющей последовательности, и в связи с этим обычный символ слэша (`'\''`) должен записываться как «двойной слэш», заключенный в апострофы (`'\\'`).

Код символа может быть непосредственно введен в строку C++. Для этого при записи строки перед шестнадцатеричным кодом должна быть записана последовательность `\x`. При использовании восьмеричного кода его признаком служит одиночный обратный слэш. Последние две строки табл. 2 демонстрируют возможность непосредственного ввода кода ASCII-символа в строку.

Таблица 2. Управляющие последовательности C++

Символ	Значение
--------	----------

<code>\a</code>	Звонок
<code>\b</code>	Удаление предыдущего символа
<code>\f</code>	Перевод формата
<code>\n</code>	Новая строка
<code>\r</code>	Возврат каретки
<code>\t</code>	Табуляция

Символ Значение

<code>\v</code>	Вертикальная табуляция
<code>\'</code>	Апостроф
<code>\"</code>	Кавычки
<code>\\</code>	Символ «обратный слэш»
<code>\0</code>	Нуль-терминатор (0)
<code>\ddd</code>	ASCII-код в 8-ричной системе счисления (здесь d — цифра от 0 до 7)
<code>\xdd</code>	ASCII-код в 16-ричной системе счисления (здесь d — цифра от 0 до 9 или буква от A до F)

unsigned char

Тип `unsigned char` используется для беззнаковых значений, находящихся в пределах от 0 до 255, и занимает один байт. Подобно типу `char` тип `unsigned char` более характерен для определения массивов, чем отдельных переменных, в частности, может быть полезен для определения буферов, хранящих последовательности двоичных данных, например:

```
unsigned char buf[1000];
```

Тип `char` также подходит для этой цели, но `unsigned char` предпочтительнее благодаря тому, что все биты этого байта равнозначны, значение числа типа `unsigned char` изменяется от 0 до 255 и является более удобным для работы с шестнадцатеричными числами от `0x0` (0) до `0xFF` (255).

signed char

Тип `signed char` используется для значений, находящихся в пределах от -128 до 127, и занимает один байт. Тип `char` часто, но не всегда, эквивалентен типу `signed char`. Подобно типам `char` и `unsigned char` тип `signed char` может задаваться для определения буферов, хранящих последовательности двоичных данных, например:

```
signed char buf[1000];
```


Типы `char`, `unsigned char` и `signed char` во многих случаях ведут себя одинаково. Разница проявляется при присваивании байтового значения более длинному целому полю (см. также раздел «Дополнительный формат» в главе «Словарь терминов и понятий»).

В следующем примере одно и то же двоичное значение 11111111 присваивается переменным, которые имеют разный тип данных, что приводит к различным результатам:

```
signed char sch;  
unsigned char uch;  
sch = 255; //все биты устанавливаются в 1,  
           //число представляется как -1  
uch = 255; //все биты устанавливаются в 1,  
           //число представляется как 255  
short i = sch; //расширяется до 1111111111111111  
short j = uch; //расширяется до 0000000011111111
```

Другое различие заключается в том, что если значение больше 128 и меньше 255 присваивается типу `signed char`, то компилятор должен его преобразовать. Возможно, при этом будет выведено предупреждение.

int

Тип `int` является наиболее используемым целым типом C и C++. Единственное, что можно сказать определенно, это то, что число типа `int` не короче числа типа `short` и не длиннее числа типа `long`:

```
int decno=40, octno=050, hexno=0x28;  
//все 3 числа принимают одно  
//и то же значение, равное 40  
int neg=-175;
```

Этот тип используется для объявления положительных или отрицательных целых чисел. В 16-разрядных системах числа типа `int` могут принимать значения от -32 768 до 32 767 и занимают 2 байта. В 32-разрядных системах числа этого типа занимают 4 байта и принимают те же значения, что и длинные целые, поэтому существует потенциальная опасность того, что в 32-разрядных системах не возникнет ошибки превышения возможного значения.

В связи с этим, несмотря на то, что тип `int` является наиболее естественным для процессора и требуется при использовании многих библиотечных функций, программисты стараются не задавать этот тип в сложных и достаточно важных программах.

unsigned int

Тип `unsigned int` имеет тот же размер, что и тип `int`: 2 байта в 16-разрядных системах (возможные значения от 0 до 65 535) и 4 байта в 32-разрядных системах (возможные значения те же, что и у `unsigned long`):

```
unsigned int decno=40, octno=050,  
hexno=0x28;  
//переменные decno, octno и hexno равны
```

Большая часть свойств, описанных для целых чисел со знаком, распространяется на беззнаковые целые — главное их отличие заключается в том, что числа типа `unsigned int` не могут принимать отрицательные значения. Для переносимости программ оптимальным решением является использование типов `unsigned short` и `unsigned long`.

Тип `unsigned int` рекомендуется для работы с полями бит, так как в этом случае они интерпретируются последовательно и являются равнозначными (см. также раздел «Операции над полями бит» в главе «Основные особенности программирования на C++»):

```
struct scard  
{  
    unsigned int field1:5;  
    unsigned int field2:2;  
    unsigned int field3:6;  
}  
cards[8];
```

Для определения константы в виде беззнакового целого можно определять ее в шестнадцатеричном виде или использовать суффикс `U`:

```
unsigned int i = 40000U;
```


short

Тип `short` определяет двухбайтные целые числа, которые могут принимать отрицательные значения. Область изменения от $-32\,768$ до $32\,767$. Этот тип более предсказуем, чем тип `int`. По умолчанию компилятор сохраняет все целые константы в виде двухбайтных целых чисел со знаком до тех пор, пока этого объема памяти достаточно для их размещения.

unsigned short

Тип `unsigned short` является двухбайтным целым, которое не может принимать отрицательные значения. Область изменения от 0 до 65 535.

Для определения константы в виде беззнакового двухбайтного числа можно определять ее в шестнадцатеричном виде или использовать суффикс `U`:

```
unsigned short i = 40000U;
```

При отсутствии суффикса, независимо от того, как это число представлено в памяти, присваивание выполнится верно:

```
unsigned short i = 40000;
```

long

Тип `long` является четырехбайтным целым типом, переменные которого могут принимать отрицательные значения. Область его изменения от $-2\,147\,483\,648$ до $2\,147\,483\,647$. Это самый «большой» стандартный целый тип ANSI C:

```
long i=100000, j=-1000000, k=-2;
```

Для представления константы в виде числа типа `long` используется суффикс `L`, в противном случае константа сохраняется в виде числа типа `short`:

```
long i=125L;
```

Использование суффикса может быть особенно полезно при выполнении арифметических действий над числами. Следующий пример демонстрирует эту особенность:


```
int a=10000;  
long res1, res2;  
res1=10*a; //неверный результат!  
res2=10L*a; //результат равен 100000
```

Первая инструкция этого примера не дает верного произведения, так как сначала выполняется умножение двух целых чисел, в результате получается целое число, которое затем присваивается переменной `res1`. Только в случае, если хотя бы один из операндов является длинным целым, в результате получается длинное целое. (Другим решением проблемы `res1` является использование приведения типа переменной `a` к типу `long`:

```
res1=10*(long)a;)
```

unsigned long

Тип `unsigned long` является четырехбайтным целым типом, переменные которого не могут принимать отрицательные значения. Область его изменения от 0 до 4 294 967 295. Отсутствие возможности для хранения отрицательных чисел компенсируется в этом типе увеличением в два раза возможных значений положительных чисел:

```
unsigned long a=500L, 4000000000L;
```

float

Тип `float` используется для четырехбайтных чисел с плавающей точкой. Область изменения чисел этого типа от $-3.4 \cdot 10^{+38}$ до $-3.4 \cdot 10^{-38}$ и от $3.4 \cdot 10^{-38}$ до $3.4 \cdot 10^{+38}$, при этом обеспечивается точность не меньше шести десятичных цифр.

Для представления чисел с плавающей точкой используются две формы записи:

- экспоненциальная, в которой запись `numEexr` означает число `num`, умноженное на 10 в степени `exr` (при этом, возможно, записи `numEexr` и `numeexr` являются идентичными);
- обычная запись в виде десятичного числа с точкой (которая заменяет привычную для математики запятую).

Для сохранения констант в виде числа с плавающей точкой обычной точности следует использовать суффикс F, в противном случае константы сохраняются в виде чисел двойной точности (компилятор при этом может выдавать предупреждение о возможной потере данных в связи с присваиванием константы двойной точности числу обычной точности):

```
float f1, f2=37.129F,  
f3=1.2e-2F, //f3=0.012  
f4=4.5E3F; //f4=4500
```



ПРИМЕЧАНИЕ При вычислении выражений, в которых участвуют числа с плавающей точкой, более предпочтительным является использование типа `double`, так как это позволяет обеспечить большую точность вычисления. Числа типа `float` чаще применяются для определения больших массивов и для сохранения информации на диске, то есть там, где критичным является объем памяти, занимаемой данными.

double

Тип `double` используется для восьмибайтных чисел с плавающей точкой. Область изменения чисел этого типа от $-1.7 \cdot 10^{+308}$ до $-1.7 \cdot 10^{-308}$ и от $1.7 \cdot 10^{-308}$ до $1.7 \cdot 10^{+308}$, при этом обеспечивается точность 15 десятичных цифр (точность вычисления и область изменения чисел этого типа зависят от версии компилятора, информация о них находится в файлах `limits.h` и `float.h`).

Для представления чисел с плавающей точкой используются две формы записи:

- экспоненциальная, в которой запись `numEexp` означает число `num`, умноженное на 10 в степени `exp` (при этом, возможно, записи `numEexp` и `numeexp` являются идентичными);
- обычная запись в виде десятичного числа с точкой (которая заменяет привычную для математики запятую).

По умолчанию все константы, не являющиеся целыми числами, сохраняются в виде чисел двойной точности:

```
double d1=0.0, d2=1.2e-125, d3=3.7E10;
```

long double

Единственное, что можно точно сказать о числах, объявленных числами этого типа, — они занимают не меньше места в памяти, чем числа типа `double`, а следовательно, обеспечивают не меньшую точность. Все остальное зависит от компилятора, поэтому этот тип следует применять только после внимательного изучения возможностей используемого компилятора, особенно в том случае, если необходимо обеспечить переносимость программного продукта.

wchar_t

Тип `wchar_t` используется для хранения символов, занимающих два байта (например, в стандарте Unicode). В связи с тем, что каждый символ этого типа занимает два байта, для формирования строк из него рекомендуются специальные функции. При отсутствии таких функций придется использовать присваивание для каждого символа:

```
wchar_t wchstr[10];  
wchstr[0]='H';  
wchstr[1]='i';  
wchstr[2]='!';
```



ПРИМЕЧАНИЕ В ранних версиях C++ этот тип не поддерживался.

Операторы

Тело программы, написанной на языке C++, состоит из последовательности инструкций (команд), каждая из которых состоит из выражений (операндов) и операторов. Операторы служат для присваивания значений переменным, реализации переходов между различными инструкциями в программе, вызова функций и т. п. Операторы могут быть унарными, бинарными или тринарными. Если операторы применяются к одному операнду — они являются унарными. Если операторы применяются к двум операндам, то они — бинарные. Условный оператор, который обозначается как `?:`, является единственным тринарным оператором — он имеет три операнда. В табл. 3 представлены операторы C++ в порядке убывания приоритета, при этом операторы с самым большим приоритетом представлены в самой верхней строке таблицы. Приоритет операторов, находящихся в одной строке, одинаков.



ВНИМАНИЕ Интерпретация унарных операций производится справа налево, бинарных — слева направо.

Таблица 3. Краткий перечень операторов

Тип оператора	Оператор
Унарный	<code>() [] -> ::.</code> <code>! ~ ++ -- - * & sizeof new delete typeid</code> приведение типа
Бинарный	<code>. * ->*</code> <code>* / %</code> <code>+ -</code> <code><< >></code>

Тип оператора	Оператор
Бинарный	< <= > >=
	== !=
	&
	^
	&&
Тринарный	
	?:
Бинарный	= += -= *= /= %= >>= <<= &= ^= =

В табл. 4 представлены операторы C++ с краткими описаниями и синтаксисом, расставленные по приоритету: от самого высокого (приоритет 1) к самому низкому (приоритет 16). Операторы с самым высоким приоритетом выполняются в первую очередь. Порядок выполнения операторов — слева направо, кроме операторов, которые имеют пометку R рядом с номером приоритета. (Например, 2R означает, что операторы второго приоритета выполняются справа налево.)

Таблица 4. Описание и синтаксис операторов

Приоритет	Оператор	Описание	Синтаксис
1	()	Вызов функции	func (args)
1	[]	Выбор элемента массива по индексу	Array [int]
1	->	Выбор члена по указателю	ptr->member
1	.	Выбор члена по имени	obj.member
1	::	Определение области видимости	class::symbol ::symbol

продолжение ↗

Таблица 4. Описание и синтаксис операторов (продолжение)

Приоритет	Оператор	Описание	Синтаксис
2R	!	Логическое НЕ	!int
2R	~	Поразрядное НЕ	~int
2R	++	Инкремент	++lval lval++
2R	--	Декремент	lval-- --lval
2R	-	Изменение знака	-num
2R	*	Обращение к памяти по значению указателя	*ptr
2R	&	Определение адреса переменной	&lval
2R	Sizeof	Определение размера операнда в байтах	sizeof (expr) sizeof (type)
2R	New	Выделение памяти для данных	new type new type (args) new type [size]
2R	Delete	Освобождение выделенной памяти	delete ptr delete [] ptr
2R	Typeid	Идентификация типа	typeid (expr)
2R	(<имя типа>)	Приведение типа	(см. главу «Операторы приведения типов»)
3	. *	Выбор члена по имени	obj.*ptr_mem
3	->*	Выбор члена по указателю	ptr->*ptr_mem

Приоритет	Оператор	Описание	Синтаксис
4	*	Умножение	num*num
4	/	Деление	num/num
4	%	Деление по модулю (остаток от деления целых чисел)	int%int
5	+	Сложение	expr + expr
5	-	Вычитание	expr - expr
6	<<	Поразрядный сдвиг влево	expr << int
6	>>	Поразрядный сдвиг вправо	expr >> int
7	<	Меньше	expr < expr
7	<=	Меньше или равно	expr <= expr
7	>	Больше	expr > expr
7	>=	Больше или равно	expr >= expr
8	==	Равно	expr == expr
8	!=	Не равно	expr != expr
9	&	Поразрядное И	int & int
10	~	Поразрядное исключающее ИЛИ	int ~ int
11		Поразрядное ИЛИ	int int
12	&&	Логическое И	expr && expr
13		Логическое ИЛИ	expr expr

продолжение ➤

Таблица 4. Описание и синтаксис операторов (продолжение)

Приоритет	Оператор	Описание	Синтаксис
14R	?:	Условный оператор	expr?expr:expr
15R	=	Присваивание	lval = expr
15R	+=	Присваивание с увеличением	lval += expr
15R	--	Присваивание с уменьшением	lval -= expr
15R	*=	Присваивание с умножением	lval *= expr
15R	/=	Присваивание с делением	lval /= expr
15R	%=	Присваивание с делением модуля	lval %= expr
15R	>>=	Присваивание со сдвигом вправо	lval >>= expr
15R	<<=	Присваивание со сдвигом влево	lval <<= expr
15R	&=	Присваивание с поразрядным И	lval &= expr
15R	^=	Присваивание с поразрядным исключающим ИЛИ	lval ^= expr
15R	=	Присваивание с поразрядным ИЛИ	lval = expr
16	,	Оператор запятая	expr1,expr2

В таблице использованы следующие обозначения:

- expr – выражение (может содержать как числа и переменные, так и адреса);
- num – число;

- `int` — выражение целого типа;
- `lval` — любое выражение, которому можно что-либо присвоить;
- `ptr` — указатель.

Ниже описаны основные операторы, которые используются в C++.

Ключевые слова — операторы

Оператор `sizeof`

Оператор `sizeof` возвращает размер в байтах для операнда, заключенного в круглые скобки. В качестве операнда может выступать либо идентификатор (переменная), либо имя типа данных.

 **ПРИМЕЧАНИЕ** Если операнд — статически распределенный массив, `sizeof` возвращает полный размер массива в байтах. Если операнд — указатель, соответствующий идентификатору массива, `sizeof` возвращает размер указателя. Если операнд — класс, структура или объединение, `sizeof` возвращает полный размер класса, структуры или объединения, включая участки памяти, используемые для выравнивания элементов.

 **ВНИМАНИЕ** Именем типа не может быть `void`.

Синтаксис:

```
sizeof (expression) // для идентификатора
sizeof (type)       // для типа
```

Рассмотрим пример:

```
// ...
// вывод на экран значения 2:
cout << sizeof (short) << '\n';
// вывод на экран значения 2 или 4:
cout << sizeof (short*) << '\n';
// вывод на экран значения 20
cout << sizeof (short [10]) << '\n';
```


Оператор new

Оператор new позволяет выделить память под динамический объект (объект, создаваемый в процессе выполнения программы). Этот оператор исполняет две функции: он автоматически возвращает указатель соответствующего типа и вызывает соответствующий конструктор, если тип данных — класс. Оператор new дает пользователю полный контроль над продолжительностью жизни данных. Если созданный объект уже не нужен, его можно удалить из памяти с помощью оператора delete.

Синтаксис:

Существует несколько форм записи оператора new:

```
new type  
new type (args)  
new type [size]
```

В каждом из нижеперечисленных случаев оператор new возвращает указатель на распределенные в памяти данные, а тип указателя представляется в виде type*:

```
int *p1, *p2;  
p1 = new int;    // выделена память под  
                // один элемент типа int  
p2 = new int [20]; // выделена память под  
                // 20 элементов типа int
```

Оператор new можно использовать, чтобы выделить память под объект типа класс, структура или объединение:

```
CAuto *pcar1, *pcar2, pcar3;  
//вызов конструктора CAuto по умолчанию:  
pcar1 = new CAuto;  
//вызов конструктора типа CAuto (char*, int):  
pcar2 = new CAuto ("Opel", 97);  
//для создания 10 объектов 10 раз  
//вызывается конструктор Cauto:  
pcar3 = new CAuto [10];
```

В третьем варианте записи оператора new аргумент size может быть числовым выражением или переменной. Таким образом, появляется возможность динамического определения массива.

Оператор delete

Оператор delete используется для удаления динамических объектов, память под которые была предварительно выделена с помощью оператора new. В качестве единственного операнда выступает указатель.

Синтаксис:

Оператор delete имеет две формы записи:

```
delete pointer;  
delete [ ] pointer;
```

Если указатель был предварительно получен с помощью оператора new type [size], то используется вторая форма оператора delete. Рассмотрим пример:

```
int *p1, *p2;  
p1 = new int;  
p2 = new int [20];  
...  
delete p1;  
delete [ ] p2;
```

Оператор typeid

Оператор typeid позволяет определять тип операнда в процессе выполнения программы (Run-Time Type Identification, RTTI). Этот оператор для обычных типов данных, как правило, не используется, но часто применяется к указателям на объекты производных классов в виртуальных функциях, реализующих полиморфизм. Для поддержки динамического полиморфизма виртуальные функции вызываются через указатель на базовый класс, который используется в качестве ссылки на объект либо базового, либо производного класса (см. также раздел «Базовые классы и указатели» в главе «Наследование»). В такой ситуации информация о типе объекта заранее неизвестна, и ее можно получить только во время выполнения программы.

Синтаксис:

При использовании оператора typeid в программу необходимо включить заголовок typeinfo.h:


```
#include <typeinfo.h>
typeid (expression | type)
```

В качестве операндов оператора `typeid` может выступать выражение `expression` или имя типа `type`. В первом случае оператор `typeid` возвращает объект типа `type_info`, который является типом выражения `expression`. Во втором случае по заданному имени типа `type` возвращается ссылка на объект типа `type_info`.

Следующий пример иллюстрирует использование оператора `typeid` для определения класса объекта, заданного с помощью указателя базового класса `pb`:

```
#include <iostream.h>
#include <typeinfo.h>
// объявление класса B и производного от него класса D
class B
{public:
    virtual int fn1(void) {return 0;}
    //...
};
class D: public B
{
    public:
    virtual int fn1 (void) { return 1;}
    virtual int fn2 (void) { return 2;}
    //...
};
// основная часть программы
void main ( )
{
    B *pb; // объявление указателя базового класса B
    D objd; // объявление элемента класса D
    pb = &objd; // присвоение переменной pb
               // адреса переменной objd
    cout << typeid (objd).name ( ) << '\n';
    cout << typeid (pb).name ( ) << '\n';
    cout << typeid (*pb).name ( ) << '\n';
}
```


В результате выполнения программы будут выведены три строки:

```
class D
class B*
class D
```

В первой из них выводится тип объекта `objd`, во второй — тип указателя на базовый класс `B` `pb`, в третьей — тип объекта `objd`, на который указывает указатель `pb`.

Тип `type_info` поддерживает два оператора: `==` и `!=`, применяемых для сравнения типов, а также две функции: `name`, которая возвращает строку, содержащую имя типа, и `before`, которая используется только для внутренних целей. Выражение `typeid(a) == typeid(b)` принимает значение **ИСТИНА**, если `a` и `b` имеют один и тот же тип. Выражение `typeid(b).name` возвращает строку, содержащую имя типа `a`.



ПРИМЕЧАНИЕ Оператор `typeid`, который не поддерживается в ранних версиях C++, позволяет пользователю получать информацию о самом объекте во время выполнения программы. Такая информация называется мета-данными и зависит от параметров компилятора (например, в Microsoft Visual C++ это ключ `/GR`).

Операторы присваивания

В C++, как и в большинстве языков программирования, поддерживается простой оператор присваивания (`=`). Этот оператор возвращает то значение, которое было присвоено. Например:

```
a = b;      // Переменной a присваивается значение b
            // и возвращается значение b
c = a = b;  // Переменной a присваивается значение b
            // и то же самое значение
            // присваивается переменной c
```

Помимо простого присваивания, C++ поддерживает большое количество сложных операторов присваивания, таких

как `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `^=`, `|=`. Каждый из таких операторов выполняет некоторую операцию с двумя операндами и затем присваивает результат левому операнду. В общем случае сложный оператор присваивания имеет следующий синтаксис:

`a @= b`

Такая запись эквивалентна следующей:

`a = a @ b`

В следующем примере две инструкции эквивалентны друг другу. В результате их выполнения сумма `x` и `y` присваивается переменной `x`:

```
x += y;  
x = x + y;
```

Поразрядные операторы

Поразрядные операторы выполняют операции над одноименными битами машинного слова. Операнды, с которыми работают поразрядные операторы, могут иметь только целочисленный тип: `int`, `short`, `long`. С любым из них рекомендуется употреблять модификатор `unsigned`. К поразрядным операторам также относятся: И (`&`), ИЛИ (`|`), исключающее ИЛИ (`^`), сдвиг влево (`<<`), сдвиг вправо (`>>`), инверсия (`~`). Все поразрядные операторы сравнивают соответствующие биты в каждой позиции машинного слова, например:

```
a соответствует 00001001 // первый операнд  
b соответствует 00000111 // второй операнд  
// Результат применения поразрядных операторов  
a & b равно 00000001  
a | b равно 00001111  
a ^ b равно 00001110
```

Оператор запятая

Оператор запятая (`,`) — простой оператор, который задает последовательность вычисления выражений и возвра-

щает значение последнего. Это полезно, когда необходимо вычислить ряд выражений в функции, обрабатывающей только одно выражение (например, проверяющей условие).

Синтаксис:

```
expression1, expression2;
```

Здесь `expression` — необходимые выражения.

В приведенном ниже примере оператор запятая используется, чтобы объединить несколько выражений в одно. Во время работы программы все они выполняются в заданной последовательности: сначала вычисляется значение переменной `a`, затем значение `b`, и лишь после этого выполняется проверка условия (`b > 0`):

```
while (a = func1( ), b = a*2, b > 0)
{
    //...тело цикла
}
```

Декремент

Оператор декремент (`--`) предназначен для уменьшения значения операнда на единицу. Оператор может стоять как перед операндом, так и после него. В первом случае сначала значение операнда уменьшается на один, затем его новое значение может быть использовано в выражении. Во втором случае сначала в выражение подставляется текущее значение операнда, а уже потом оно уменьшается на единицу.

Пусть `b = 3`, `c = 7`, тогда

```
a = b + c--; // Результат: a=10, b=3, c=6
```

```
a = b + --c; // Результат: a=9, b=3, c=6
```

Инкремент

Оператор инкремент (`++`) предназначен для увеличения значения операнда на единицу. Оператор может стоять как

перед операндом, так и после него. В первом случае сначала значение операнда увеличивается на единицу, и его новое значение используется в дальнейших вычислениях. Во втором случае сначала подставляется текущее значение операнда, а уже потом оно увеличивается на единицу.

Пусть $b = 3$, $c = 7$, тогда

```
a = b + c++; // Результат: a=10, b=3, c=8
```

```
a = b + ++c; // Результат: a=11, b=3, c=8
```

Условный оператор

Условный оператор (`?:`) представляет собой компактную форму инструкции условия `if`. Это единственный тринарный оператор в языке C++.

Синтаксис:

```
test_expression ? true_expression : false_expression
```

Выполнение этой инструкции начинается с операнда `test_expression`. Если значение выражения `test_expression` есть ИСТИНА, то возвращается значение выражения `true_expression`, иначе возвращается значение выражения `false_expression`.

В приведенном ниже примере выполняется выбор наибольшего из двух значений. Если значение `a` больше `b`, то возвращается `a`, иначе возвращается `b`:

```
max = (a > b ? a : b);
```

Логические операторы

К логическим операторам относятся операторы: конъюнкции (логическое И), дизъюнкции (логическое ИЛИ) и логического отрицания. В отличие от поразрядных операторов логические операторы сначала приводят свои операнды к значению ИСТИНА или ЛОЖЬ. Как уже говорилось, при этом все значения, отличные от нуля, интерпретируются как ИСТИНА, а ноль (0) интерпретируется как ЛОЖЬ. Кроме того, двоичные логические операторы (И и ИЛИ)

используют так называемую «короткую схему» логики: если значения уже вычисленной части выражения достаточно для определения результата всего выражения, вычисление этого выражения прекращается. Такой подход также называют сокращенным вычислением логических выражений (его применение зависит от установленных параметров компилятора). В следующем примере используется «короткая схема» логики. То есть в данном конкретном случае для оператора И (&&) если первый операнд есть ЛОЖЬ, то второй операнд обрабатываться не будет:

```
if (a > b && func1( ) >= c)
```

...

Оператор деления по модулю

Оператор деления по модулю (%) возвращает остаток от деления двух целых чисел. Наиболее часто этот оператор используется для того, чтобы определить, делится ли одно целое число нацело на другое; и если делится, то возвращается 0.

Например, функция `func_even` возвращает значение ИСТИНА, если параметр `n` — четный:

```
bool func_even (int n)
{
    return (n % 2) == 0;
}
```

Операторы-указатели

В C++ существуют операторы, которые имеют дело с получением и использованием адресов. Адреса в свою очередь могут быть сохранены в переменных, называемых указателями (см. также главу «Указатели, массивы и строки»). Синтаксис и краткое описание операторов для работы с указателями приведены в табл. 5. В данной таблице `ptr` — это любое имеющее силу выражение для вычисления адреса.

Таблица 5. Операторы для работы с указателями

Синтаксис	Описание
<code>*ptr</code>	Значение, адрес которого задан указателем <code>ptr</code>
<code>&lvalue</code>	Адрес переменной <code>lvalue</code>
<code>ptr->member</code>	Значение члена <code>member</code> (например функции) — объект, адрес которого задается указателем <code>ptr</code> . Эта запись эквивалентна инструкции <code>(*ptr).member</code>
<code>ptr[n]</code>	Имя массива является указателем на массив, поэтому запись <code>ptr[n]</code> эквивалентна записи <code>*(ptr + n)</code> и определяет значение <code>n</code> -го элемента массива

Символы `*` и `&` в данном случае являются унарными операторами для работы с адресами. (Их нельзя путать с операторами умножения и поразрядного И.)

Таблица 6. Специальные операторы

Синтаксис	Описание
<code>object.*ptr_to_member</code>	Доступ члена объекта <code>object</code> определяется по смещению, задаваемому указателем <code>ptr_to_member</code>
<code>ptr -> *ptr_to_member</code>	Сначала через указатель <code>ptr</code> определяется объект, затем по смещению <code>ptr_to_member</code> определяется нужный член объекта

Операторы-указатели на член объекта

В C++ существует два довольно редко используемых специальных оператора, называемых указателями на член

объекта. Эти операторы указываются с переменной, объявляемой с именем класса, к которому относится объект, но членом класса не являются:

```
<тип> <имя класса>::*ptr_to_member;
```

Такая переменная фактически не является указателем, но она задает смещение внутри класса (объекта). Следует отметить, что во время выполнения программы эта переменная может указывать на разные члены объекта.

Синтаксис и краткая характеристика таких операторов представлены в табл. 6.

Например, объявим объект car класса Ccar:

```
class CCar
{
    int speed;
    int power;
    int age;
    int f1 (void);
}
car;
//...
int CCar::*pmem;
//объявление переменной-указателя на член объекта:
//...
pmem = &CCar::power;
//определение смещения для члена power
int i = car.*pmem; //эквивалентно int i=car.power;
//...
pmem = &CCar::age;
//определение смещения для члена age
int j = car.*pmem; //эквивалентно int j=car.age;
```

Операторы отношения

Операторы отношения сравнивают первый операнд со вторым и возвращают единицу, если результат ИСТИНА, и ноль — если результат ЛОЖЬ. К таким операторам относятся операторы: больше (>), меньше (<), больше либо

равно ($\geq$), меньше либо равно ($\leq$), тождественно равно ($\equiv$), не равно ($\neq$). Все операторы отношения относятся к бинарным операторам. В качестве операндов могут быть использованы любые выражения целого типа или типа с плавающей точкой, а также выражения типа указатель.

Оператор определения области видимости

Оператор определения области видимости ($::$) имеет очень высокий приоритет, потому что имя объекта должно быть определено с самого начала. В C++ имеется возможность объявить член класса и использовать его в любом другом месте программы. Это возможно с помощью оператора определения области видимости ($::$).

Синтаксис:

`class::symbol`

ссылка на член класса или

`::symbol`

ссылка на глобальную переменную.

В приведенном ниже примере выполняется обращение к функции `go_faster` как к элементу класса `CAuto`. Переменная `speed` так же объявлена, как элемент класса:

```
class CAuto
{
    private:
        double speed;
    public:
        double go_faster (double inc);
        //...
};
CAuto::go_faster (double inc)
{
    speed += inc;
    return speed;
}
//...
```


Вторая форма оператора полезна, когда необходимо использовать переменную как глобальную, а не как член класса. (По умолчанию, если имеется одноименные элементы в пределах класса и в программе, компилятор интерпретирует элемент скорее как член класса, чем как глобальную переменную.) В данном примере объявлены две переменные `speed`: одна как член класса `CAuto`, вторая как глобальная переменная. Компилятор интерпретирует их как две разные переменные:

```
double speed;  
//объявление глобальной переменной speed  
//вне класса CAuto  
CAuto::go_faster (double inc)  
{  
    speed += inc;  
    ::speed = speed; // присвоить глобальной переменной  
                     // значение элемента класса  
    return speed;  
}
```


Операторы приведения типов

Операторы приведения типов (cast operators) позволяют изменять тип выражения: они вычисляют значение выражения, изменяют его тип и присваивают значение нового типа результату. В некоторых случаях это может привести к изменению внутреннего формата данных (изменению вида данных в памяти компьютера). Ниже приведены основные операторы приведения типов и способы их использования.

- `const_cast`. Позволяет удалить модификатор `const` из указателя. Этот оператор предоставляет возможность передавать указатель, объявленный с модификатором `const`, аргументу, объявленному без этого модификатора (см. раздел «`const`» в главе «Ключевые слова»).
- `dynamic_cast`. Во время выполнения программы проверяет, что объект, на который ссылается указатель, имеет определенный тип.
- `reinterpret_cast`. Выполняет преобразование указателей на пустой тип (`void*`) в указатели на более конкретные типы (`int*`, `char*`, `float*` и т. д.).
- `static_cast`. Обеспечивает преобразование родственных типов указателей или объектов. Этот оператор можно использовать для подавления выдачи предупреждений при компиляции в случае замены типа, занимающего больший объем памяти, на тип, занимающий меньший объем памяти.

Операторы приведения типов языка C поддерживают все эти операции, однако синтаксис в этом случае является другим. Оператор приведения `dynamic_cast` является новым элементом ANSI C++.

Таким образом, ANSI C++ предоставляет четыре разных оператора приведения, каждый из которых обладает специ-

фическими возможностями, описанными выше. Операторы C оставлены в C++ для обеспечения преемственности.

Оператор приведения типа старого стиля

Язык C и ранние версии C++ предоставляют всего один оператор приведения типов практически для всех ситуаций. В ANSI C++ этот важный оператор приведения также сохранен.

Синтаксис:

Следующее выражение имеет то же значение, что и выражение `expr`, но оно приведено к типу, определенному в скобках:

```
(type) expr
```

Хотя выражение `(type) expr` теоретически имеет то же значение, что и выражение `expr`, действие оператора приведения типа может привести к преобразованию данных, которые изменят внутреннее представление числа в памяти. Например, целое число 10 и десятичное 10.0 будут по-разному представлены в двоичной системе (именно она используется в компьютере).

Оператор приведения типа старого стиля также можно определить, используя альтернативный синтаксис:

```
type(expr)
```

Приведенный ниже пример позволяет вывести на печать текущее значение целого `i` в формате с десятичной точкой:

```
int i = 11;  
cout << (double) i;
```

Последняя инструкция эквивалентна инструкции

```
cout << 11.0;
```

В целом существует очень мало ситуаций, когда оператор приведения типов может как-то повлиять на поведение программы. Одна из таких ситуаций показана в следующем примере. Присваивая значения и вызывая функции, компилятор автоматически переводит целые числа в деся-

тичные, поэтому в таком случае явное выполнение приведения типа не нужно. Например, в следующем фрагменте программы приведение переменной `i` к типу `double` является лишним:

```
int i = 5;  
double x = (double) i;
```



ПРИМЕЧАНИЕ Операторы приведения типов языка C можно использовать во всех ситуациях, в которых используются операторы приведения типов ANSI C++. Исключение составляет только новый, отсутствующий в C оператор `dynamic_cast`.

Примером к этому примечанию могут служить два следующих оператора, выполняющие одни и те же действия:

```
char *p = (char *) malloc(n);  
char *p = reinterpret_cast<char*>(malloc(n));
```

Следующая пара операторов, которые подавляют предупреждение компилятора, также выполняют одно и то же действие:

```
bool flag1 = (bool) 12;  
bool flag1 = static_cast<bool>(12);
```

Удаление модификатора `const`

Оператор приведения типа `const_cast` предусмотрен для облегчения вызова функций, которые должны использовать модификатор `const`, но не используют его. Такие функции работают с параметром типа указатель и никогда не изменяют данные, на которые ссылается этот указатель, даже если программист не определил параметр как `const`. Правила C++ запрещают передачу константного указателя в качестве фактического параметра такой функции, оператор `const_cast` предоставляет возможность обойти этот запрет. (См. также раздел «`const`» в главе «Ключевые слова».)

Синтаксис:

Следующее выражение имеет то же значение, что и `expr`.
Обозначенный тип `type` должен быть таким же, что и тип

expr, кроме отсутствия модификатора const или volatile. Как правило, type — это тип указателя:

```
const_cast<type> (expr)
```

В следующем примере функция display_num использует в качестве параметра указатель, но не данные, на которые он указывает (*p):

```
void display_num(double *p)
{
    printf ( "The value is %6.3f/n", *p);
}
```

Так как заданные посредством указателя p данные не изменяются, должна быть возможность передачи в качестве параметра const-указателя. Правила C++ запрещают это из-за несоответствия типов указателей:

```
const double x;
display_num(&x); // Ошибка, так как переменная &x
                // объявлена как const
```

Чтобы обойти этот запрет, нужно лишить указатель модификатора const. Следующая операция позволяет привести &x от const double* к double* и таким образом устранить запрет:

```
const double x = 11.3;
display_num(const_cast<double*>(&x));
```

 **ВНИМАНИЕ** При использовании оператора const_cast необходимо быть уверенным, что данные, на которые указывает указатель, не изменяются. При задании оператора const_cast и одновременной попытке изменить данные результаты выполнения программы будут непредсказуемы.

```
const double x = 17.5;
double *p;
P = const_cast<double*>(&x);
*P = 33.0; //этот оператор не определен
```

 **ПРИМЕЧАНИЕ** Оператор const_cast не поддерживался в ранних версиях C++.

Проверка типа во время выполнения программы

Оператор `dynamic_cast` предоставляет возможность проверить, что указатель базового класса указывает на объект определенного производного класса. Оператор `dynamic_cast` проверяет тип объекта во время выполнения программы, используя информацию RTTI (run-time type information).

Синтаксис:

Следующее выражение должно возвращать ссылку на заданный тип `type`. Во время выполнения задачи объект, указанный в аргументе `expr`, должен иметь определенный тип `type` или тип, производный от типа `type`, иначе приведение типа не выполняется и возвращается нулевой указатель:

```
dynamic_cast<type*>(expr)
```

Оператор `dynamic_cast` также можно использовать со ссылочными типами. Если приведение типа не выполняется из-за того, что `expr` имеет тип, не совпадающий с типом `type` или типом, производным от `type`, то вырабатывается исключение `bad_cast`.

Для этого оператора приведения типов существует единственное ограничение: при использовании его для приведения к указателю на производный класс (что является его главной целью) класс выражения `expr` должен иметь, по крайней мере, одну виртуальную функцию. По этой причине оператор `dynamic_cast` часто называется оператором *полиморфного приведения типа* (polymorphic cast). Также возможно применение оператора `dynamic_cast` для приведения к указателю базового класса без каких-либо ограничений. Если типы не связаны вообще, компилятор не допускает преобразования.

Наиболее полезным в операторе `dynamic_cast` является то, что указатель базового класса может указывать на производные классы. Некоторые из этих классов поддерживают функции, которых нет в других классах.

Например, следующий фрагмент программы определяет базовый класс B и производный от него класс D, а затем присваивает его адрес указателю базового класса:

```
class B
{public:
    virtual void func1(int);
};
class D : public B
{public:
    void func2(void);
};
//...
D od;
B *pb
pb = &od;
```

В последней строке этого примера указателю на класс B присваивается адрес объекта производного класса D.

Следующая функция использует оператор `dynamic_cast`, чтобы проверить, действительно ли параметр типа `B*` указывает на объект класса D. Если это так, то вызывается функция `func2`, которая определена только в классе D и классах, производных от D:

```
void process_B (B* arg)
{
    D *pd;
    pd = dynamic_cast<D*>(arg);
    if (pd) pd->func2();
//...
}
```

Если `arg` указывает на объект класса D или класса, производного от D, то преобразование типа будет выполнено, и функция `process_B` вызовет функцию `D::func2` через указатель `pd`. Иначе операция преобразования не выполняется и `pd` присваивается значение `NULL`:

```
pd = dynamic_cast<D*>(arg);
```




ПРИМЕЧАНИЕ Оператор `dynamic_cast` не поддерживался в ранних версиях C++. Некоторые компиляторы предоставляют это средство только при включении определенного ключа компилятора, например `/GR` для Microsoft Visual C++ 6.0.

Приведение типа указателя

Оператор `reinterpret_cast` используется для приведения типа указателя к другому типу. Новый тип не обязательно должен быть связан со старым, этот же оператор позволяет приводить типы между указателями и целыми числами.

Синтаксис:

Следующий оператор возвращает то же значение выражения, что и выражение `expr`, но переведенное к заданному типу `type`. Этот тип, который обычно является типом указателя, может отличаться от типа выражения `expr` только интерпретацией (способом использования). Например, указатель `float*` используется иначе, чем указатель `char*`. Сами данные и их внутреннее представление в памяти при этом не изменяются:

```
reinterpret_cast <type> (expr)
```

Этот оператор нельзя использовать для удаления модификаторов `const` или `volatile` из выражения `expr`, так как это могло бы привести к возможности приведения типов с помощью оператора `reinterpret_cast` между указателями, а также между указателями и целыми числами без каких-либо ограничений.

Вообще оператор `reinterpret_cast` никогда не изменяет внутреннее представление данных, на которые ссылается указатель, определяемый выражением `expr`. Тем не менее, когда тип указателя и тип данных, на которые он указывает, не соответствуют друг другу (например, для указателя используется тип `int*` вместо `float*`), результат будет просто драматичен.

Наиболее часто этот оператор используется для преобразования значения, возвращаемого как `void*`, к более конкретному типу. В C++ нельзя присвоить указатель, объявленный как `void*`, указателю другого типа без приведения типа. Например:

```
char *p = reinterpret_cast<char*>(malloc(100));
```

Хотя возвращенное значение функции `malloc` не обязательно переводить в другой тип указателя немедленно, такое приведение должно быть выполнено перед первым обращением к выделенной области памяти. Одним из преимуществ оператора `new` перед оператором `malloc` является отсутствие необходимости приведения типов.

Оператор `reinterpret_cast` может быть также полезен при определении функции, когда функция получает указатель типа `void*` в качестве параметра (см. также раздел «`qsort`» в главе «Библиотечные функции»).



ВНИМАНИЕ Использование оператора `reinterpret_cast` требует большой аккуратности. Например, если перевести указатель типа `int*` в указатель типа `float*`, то целые числа, обращение к которым будет производиться через указатель на тип `float`, будут считываться некорректно.

Рассмотрим пример:

```
int i = 100;  
int *pi = &i;  
float *pf = reinterpret_cast<float*>(pi);  
cout << *pf; //неверный вывод чисел
```

Последствия приведения типов указателей с помощью оператора `reinterpret_cast` очень сильно отличаются от последствий приведения типов объектов, выполняемого оператором `static_cast`. В приведенном выше примере компилятором будет выдано сообщение `Attempt to read integer data as if it were in floating-point format` (Попытка прочитать целые числа в формате чисел с плавающей запятой).

Использование оператора `static_cast <float>`, напротив, преобразует целое число в формат числа с плавающей точкой:


```
int = 100;  
float f = static_cast<float>(i) ;  
cout << f;
```



ПРИМЕЧАНИЕ В ранних версиях C++ оператор `reinterpret_cast` не поддерживался.

Преобразование типа между родственными объектами или указателями

Оператор `static_cast` — это оператор приведения типов между родственными объектами или типами указателей. Участвующие в этой операции классы должны быть связаны через наследственность, конструктор или функцию преобразования. Оператор `static_cast` также работает с простейшими числовыми типами данных.

Синтаксис:

Следующее выражение имеет то же значение, что и выражение `expr`, но позволяет переводить его в тип `type`:

```
static_cast <type> (expr)
```

Хотя численное значение результата не изменяется, при выполнении этого оператора может быть изменено внутреннее представление данных. Например, целое число 10 и десятичное 10.0 по-разному представляются в памяти компьютера.

Оператор `static_cast` используется в следующих случаях:

- для приведения типов между родственными типами указателей, если они указывают на классы, которые связаны через наследственность. В отличие от `dynamic_cast` можно выполнять приведение к типу указателя на производный класс даже при отсутствии виртуальных функций, проверка во время выполнения программы также выполняется;
- приведение к простейшим типам данных;

- приведение выражения типа А к типу В, когда В имеет соответствующий конструктор из типа А или А имеет функцию преобразования в В.

В принципе, можно использовать оператор `static_cast` всегда, когда допускается автоматическое преобразование в обратном направлении. Например, целые числа автоматически переводятся в десятичные в выражении, подобном `f = i`. Чтобы выполнить обратное преобразование, требуется оператор `static_cast` (`i = f`).

В основном оператор `static_cast` используется для приведения сложных типов к простым: при работе с простыми типами он подавляет предупреждение компилятора:

```
long j = 17;  
short i = static_cast<short>(j);
```

Этим вы как бы говорите: «Да, я на самом деле хочу это сделать». И компилятор вежливо вам потакает (иначе появляется предупреждение о возможной потере данных). При этом необходимо быть уверенным в том, что данные не будут слишком велики для приведения в другой тип, в последнем случае в `short`.

Таким же образом возможно преобразование типов из указателя базового класса в производный без каких-либо запретов. Но так как никакой проверки во время выполнения программы не делается, то забота о поддержке данных в должном виде возлагается на программиста (производный класс содержит все члены базового класса плюс члены производного класса, значения последних в этой ситуации должны быть в явном виде определены в программе). Например, в следующем примере В является базовым классом класса D:

```
B *pb;  
//...  
D *pd;  
pd = static_cast<D*>(pb);
```

В данном случае объект, на который указывает `pb`, фактически имеет тип D или производный от него. Однако при

этом значения всех членов класса D, не входящих в класс B, не определены, и обращение к ним приведет к возникновению ошибок. В такой ситуации программист сам должен определить все недостающие члены с помощью присваивания (см. также раздел «Проверка типа во время выполнения программы», в котором также описана возможность приведения типа к типу указателя производного класса). Конечно, можно пойти в обратном направлении — выполнить присваивание указателю на базовый класс. В этом случае оператор приведения типа является лишним и оператор

```
pb = static_cast<*B>(pd);
```

эквивалентен оператору

```
pb = pd;
```

Другие случаи, в которых оператор `static_cast` может быть полезен, редко встречаются в программах. Например, с его помощью можно последовательно выполнить несколько преобразований типов данных. В следующем примере A и B — такие классы, в которых A имеет функцию преобразования в B, а B — функцию преобразования к типу `int`. Тогда объект типа A может быть приведен к типу `int` следующим образом:

```
A oa;
```

```
int i = static_cast<int>(static_cast<B>(oa));
```

Без использования этой цепочки преобразований (и при отсутствии функции преобразования A в тип `int`) преобразование `oa` в целое было бы невозможным.



ПРИМЕЧАНИЕ Такую последовательность при приведении типов можно записать с помощью оператора приведения типов языка C: `int i = (int) (B) oa;`.

На первый взгляд кажется, что при использовании оператора `static_cast` программисту придется выполнить больше работы, чем при программировании старыми методами. Однако при этом становятся доступными возможности

ANSI C++ по разделению операторов приведения типа, что значительно облегчает просмотр больших программ.

Оператор `static_cast` также полезен при применении различных форматов данных или при необходимости вывода данных в формате, отличном от того, который определен для данного типа по умолчанию. Например, нужно вывести целое число `i` в формате с плавающей точкой:

```
int i = 25;  
cout << static_cast<double>(i);
```

Заметим, что для вывода на экран адреса строки `psame` надо использовать оператор `reinterpret_cast`, так как имя строки имеет тип указателя, а не простейший тип:

```
char str[] = "Hello";  
cout << hex << reinterpret_cast<int>(str);
```



ПРИМЕЧАНИЕ Оператор `static_cast` не поддерживался в ранних версиях C++.

Ключевые слова

Ключевые слова — это универсальные слова, используемые при программировании на языке C++. В C и C++ ключевые слова не включаются в библиотеки.

asm

Заголовок `asm` используется для встраивания команд на языке ассемблера в программу на C++.

auto

Спецификатор `auto` используется для объявления локальных переменных, однако, поскольку переменные являются локальными по умолчанию, данное ключевое слово встречается редко.

Например, переменная `data_declaration` объявляется как локальная:

```
auto data_declaration;
```

break

Инструкция `break` применяется для немедленного выхода из инструкций цикла `for` и `while`, а также инструкции выбора `switch`. При вложении циклов инструкция `break` позволяет выйти из тела самой внутренней инструкции цикла. После выполнения инструкции `break` управление будет передано первой инструкции, расположенной следом за последней инструкцией прерванного цикла.

case

Заголовок `case` применяется в инструкции выбора `switch` для определения одного из выбираемых вариантов.

Синтаксис:

```
case constant_value:
    statements;
```

(См. также раздел «switch».)

В приведенном ниже примере в блоке инструкции switch управление передается той инструкции case, аргумент constant_value которой равен заданному в switch аргументу num. Если num = 1, на экран будет выведено слово «Один», если num = 2 — слово «Два»:

```
switch (num)
{
    case 1:
        printf ("Один \n");
        break;
    case 2:
        printf ("Два \n");
        break;
}
```

catch

Инструкция catch может являться частью механизма обработки исключений try или определять обработчик исключений. Причем в структуре try может быть несколько блоков инструкций catch, каждый из которых предназначен для обработки определенного типа исключений. (См. также раздел «try».)

Синтаксис:

```
catch (exception_type [object])
{
    statements;
}
```

ИЛИ

```
catch (...)
{
    statements;
}
```


Аргумент `exception_type` — тип исключения, перехватываемый инструкцией `catch`. Инструкция `catch (...)` перехватывает исключения всех типов. Необязательный аргумент `object` определяет исключение с именем `object` — его можно использовать в теле блока `catch`.

class

Спецификатор типа `class` определяет имя класса объектов.

Синтаксис:

```
class class_name [: base_class_declarations]
{
    declarations;
}
[object_definitions];
```

Аргументы:

- `class_name` — имя класса, необязательный аргумент;
- `base_class_declarations` — имя базового класса, от которого образован класс `class_name`, если он является производным;
- `declarations` включает в себя как объявления переменных и функций, так и способы выделения памяти для них.

По умолчанию устанавливается метод доступа `private` (см. описание модификаторов доступа). В объявлении класса список переменных `object_definitions` типа `class_name` может отсутствовать.

В приведенном ниже примере объявляется класс `CIOStr`, полученный из базового класса `CStr`. Кроме того, объявляется несколько объектов класса `CIOStr`: `str1`, `str2`, `str3`. Обратите внимание, что объявленная в данном классе переменная `status` по умолчанию имеет тип доступа `private`, а объявленные ниже функции `input` и `output` — тип доступа `public`:

```
class CIOStr: CStr
{
    int status;
```



```
public:
void input (void);
void output (void);
}
str1, str2, str3;
```

const

Модификатор `const` определяет переменную, значение которой не должно изменяться (так называемые константные переменные, или символические константы).

Синтаксис:

```
const type item;
```

Если модификатор `const` указан в начале объявления переменной, она не может быть изменена в дальнейшем, при этом переменная должна быть инициализирована:

```
const int i = 5; // переменная i объявлена как константа
```

Можно объявлять указатель на константу, при этом находящийся в указателе адрес может измениться, а содержимое ячейки памяти по этому адресу — нет:

```
const type *pointer;
```

Можно объявлять так называемый константный указатель, при этом адрес меняться не может, а содержимое памяти по этому адресу — может. В этом случае синтаксис будет видоизменен.

Синтаксис:

```
type *const pointer;
```

Пример:

```
int *const p = &i; // указатель p объявлен как константный
```

Можно объявлять константный указатель на константу, при этом ни адрес, ни содержимое по этому адресу не могут быть изменены.

Синтаксис:

```
const type *const pointer;
```


Пример:

```
const int *const p = &i;
```

continue

Инструкция `continue` используется для игнорирования оставшихся инструкций в теле цикла и перехода к очередной проверке условия выполнения тела цикла. То есть в отличие от инструкции `break` управление передается не на следующую после цикла инструкцию, а на следующий шаг итерации.

В приведенном ниже примере осуществляется вывод четных чисел. Когда переменная `number` становится нечетной, выполняется инструкция `continue`, которая передает управление на следующий шаг цикла, не выполняя функцию `printf`:

```
for (number = 0; number < 50; number++)  
{  
    if (number % 2) continue;  
    printf ("% d", number);  
}
```

default

Заголовок `default` используется в инструкции `switch` для указания инструкций, выполняющихся в случае, если проверяемое значение не совпадает ни с одним из значений, стоящих после инструкции `case`. (См. также раздел «switch».)

Синтаксис:

```
default: statements;
```

do

Инструкция `do` является началом инструкций цикла с постусловием. Так как проверка условия выполняется после выполнения тела цикла, эта инструкция выполняется не менее одного раза.

Синтаксис:

```
do
{
    statements;
}
while (condition);
```

Аргументы:

- `statements` — инструкции, выполняемые в теле цикла;
- `condition` — выражение, определяющее условие повторения тела цикла.

Цикл выполняется до тех пор, пока значение аргумента `condition` является истиной (TRUE). Если в теле цикла указана только одна инструкция, фигурные скобки можно опустить.

else

Инструкция `else` является необязательной частью условной инструкции `if`. (См. также раздел «if».)

enum

Спецификатор `enum` используется для определения перечисляемых типов данных. Эти типы могут быть полезны при задании списка констант, все элементы которого различны.

Синтаксис:

```
enum [enum_type_name]
{
    item_1, item_2, ..., item_n
} [variable_declarations];
```

Аргументы:

- `item_1, item_2, ..., item_n` — последовательность элементов, в которой компилятор ставит в соответствие каждому элементу номер. По умолчанию нумерация элементов начинается с нуля, и каждый следующий элемент

имеет номер на единицу больше номера предыдущего элемента. При перечислении это позволяет задавать номера элементов в явном виде;

- `enum_type_name` — необязательный аргумент, характеризующий имя перечисляемого типа.

Следующая структура создает тип `card_suit`, в котором переменные `card1` и `card2` объявлены как переменные типа `card_suit`, и `pack[52]` — массив типа `card_suit`:

```
enum card_suit
{
    CLUBS,
    DIAMONDS,
    HEARTS,
    SPADES
} card1;
card_suit card2, pack [52];
```

explicit

Одним из преимуществ конструкторов является то, что они автоматически выполняют создание объекта определенного класса (см. также раздел «Перегрузка конструкторов» в главе «Конструкторы»). Модификатор `explicit` позволяет предотвратить такое создание и используется крайне редко.

Синтаксис:

```
class MyClass
{
    //...
    //определение конструктора класса MyClass,
    // создающего объект этого класса
    // из переменной v типа MyType:
    explicit MyClass (MyType *v);
    //...}
    {...}
```



ПРИМЕЧАНИЕ

В ранних версиях C++ модификатор `explicit` не поддерживался.

extern

Модификатор `extern` описывает переменную, объявленную в другом файле или другом месте программы (такая переменная называется внешней). Модификатор `extern` обычно используется при раздельной компиляции исходных файлов, в дальнейшем объединяемых при построении исполняемого (.exe) файла.

Синтаксис:

```
extern ["C"] declaration;
```

Аргументы:

- `declaration` — объявления переменных;
- `"C"` — необязательный параметр, указывающий, что элементы объявлены, используя соглашения языка C. Эта особенность дает возможность обратиться к элементам в модуле языка C без перезаписи и компиляции его как модуля языка C++.

for

Инструкция цикла `for` позволяет осуществить проверку условия выполнения цикла, инициализировать и изменять переменные.

Синтаксис:

```
for (initializer; condition; modification)
{
    statements;
}
```

Аргументы:

- `initializer` — список инициализирующих выражений или объявлений переменных с инициализацией;
- `condition` — условие повторения цикла;
- `modification` — список выражений, изменяющих значение переменных;
- `statements` — инструкции, выполняемые в теле цикла.

Цикл будет выполняться до тех пор, пока значение выражения `condition` будет истинным (`TRUE`). Такая инструкция эквивалентна следующей последовательности:

```
initializer; //set i = 1;
while (condition) //while ( i<=100)
{
    statements;
    modification; //i++;
}
```

В приведенном ниже примере выполняется вывод на экран последовательности целых чисел от 0 до 50:

```
for (i = 0; i <= 50; i++)
{
    printf ("%d \n", i);
}
```

friend

Спецификатор доступа `friend` используется для объявления функций или классов, дружественных объявленному классу. Дружественные функции или классы не являются членами класса, но имеют доступ к закрытым (`private`) и защищенным (`protected`) членам класса. Главной целью создания друзей классов является возможность перегрузки с их помощью функций бинарных операторов (см. также раздел «Для чего нужны функции-друзья» в главе «Перегрузка операторов»).

Функции-друзья не могут быть вызваны по указателю `this` или применяться к объектам класса, в котором они объявлены со спецификатором `friend`, с помощью операторов доступа к членам класса (`.` и `->`).



ВНИМАНИЕ Дружественные функции или классы принято объявлять перед членами класса и перед ключевыми словами `public`, `protected`, `private`, так как на них не действуют обычные атрибуты видимости: класс или функция либо являются дружественными, либо нет.

В приведенном ниже примере класс CarCar объявляется дружественным классу Car.

```
class Car
{
    friend class CarCar;
public:
    int C;
    //...
}
```

 **ПРИМЕЧАНИЕ** Если класс C1 дружествен классу C2, а C2 дружествен C3, C1 не становится автоматически дружественным классу C3, то есть «друг моего друга не обязательно мой друг». Если класс C1 дружествен классу C2, то класс D, производный от класса C1, не становится автоматически дружественным классу C2, то есть «ребенок моего друга не обязательно мой друг».

goto

Инструкция безусловного перехода goto осуществляет переход в процессе выполнения программы к метке, имя которой указано после ключевого слова goto. Однако использование этой инструкции в C и C++ не приветствуется, так как нарушает логическую структуру программы.

Синтаксис:

```
goto label;
```

Здесь label — имя метки.

В представленном ниже примере после выполнения инструкции goto управление передается на инструкцию return с меткой the_end:

```
goto the_end;
//...
the_end: return x;
```

if

Инструкция if — первая часть инструкции ветвления. (См. также раздел «else».)

Синтаксис:

```
if (condition)
{
    bloc_1_statements;
}
else
{
    bloc_2_statements;
}
```

Аргументы:

- `condition` — выражение, определяющее, какой из двух блоков будет выполнен;
- `bloc_1_statements`, `bloc_2_statements` — инструкции первого и второго блоков соответственно.

Если значение выражения `condition` — истина (`TRUE`), то выполняются инструкции первого блока, иначе — второго (если он имеется). Вторая часть условной инструкции, начинающаяся со слова `else`, не обязательна. Если после инструкции `if` или `else` стоит всего одна инструкция, фигурные скобки можно опустить.

Пример:

```
if (x < y)
    puts ("x меньше y");
else
    if (x > y)
        puts ("x больше y");
    else
        puts ("x равно y");
```

inline

Модификатор `inline` используется для описания встраиваемых функций. Такие функции уменьшают число операций процессора, выполняемых при их вызове, в связи с чем повышается быстродействие программ. Однако при этом увеличивается размер программ, поэтому встраиваемыми следует определять только очень короткие функции, состоящие не более чем из двух-трех инструкций.

В приведенном ниже примере описывается встраиваемая функция `cube`, возвращающая куб числа `x`:

```
inline double cube (double x)
{
    return x*x*x;
}
```

main

Функция `main` — функция, с которой начинается выполнение любой стандартной программы на C++. Функция `main` может возвращать значения типа `int` или `void`.

Например, представленная ниже функция не возвращает никакого значения:

```
void main ( )
{
    init_vars ( );
    prompt_for_input ( );
    print_results ( );
}
```

Следующая функция возвращает значение типа `int`:

```
int main (int argc, char *argv [ ])
{
    if (argc != 3)
    {
        printf ("Bad number of arguments.\n");
        printf ("Syntax: sort infile outfile.\n");
        return -1;
    }
    //...
}
```

mutable

Модификатор `mutable` применяется для того, чтобы обеспечить возможность изменения отдельных данных-членов класса в том случае, если этот класс объявлен как `const`.



ПРИМЕЧАНИЕ В ранних версиях C++ модификатор `mutable` не поддерживался.

namespace

Спецификатор `namespace` определяет пространство имен функций, классов и переменных, находящихся в отдельной области видимости. Вообще `namespace` определяет группу символов, представляющих собой названия или имена функций, классов и переменных. Примером такого определения области видимости является область видимости членов класса. (См. также раздел «`using`».)

Синтаксис:

```
namespace name
{
    // объявления и определения имен:
    declarations;
}
```

Аргументом спецификатора `namespace` (то есть `name`) является идентификатор пространства имен.

Чтобы обратиться к чему-нибудь в пространстве `namespace_name`, достаточно обратиться непосредственно к члену, используя оператор определения области видимости (`::`):

```
namespace_name::member;
```

Однако при многократном использовании члена `member` такая запись становится слишком громоздкой. Инструкция `using` заставляет компилятор признавать дальнейшее использование этому члену пространства `namespace_name` без дополнительного определения имени этого пространства:

```
using namespace_name::member;
```

Инструкция `using namespace` заставляет компилятор признавать все члены пространства имен `namespace_name`:

```
using namespace namespace_name;
```


В приведенном ниже примере глобальные переменные Cary и Hugh объявляются в пространстве имен grants:

```
namespace grants
{
    int Cary = 0;
    int Hugh = 1;
}
```

Вне пространства имен grants переменные могут быть упомянуты как grants::Cary и grants::Hugh. Но можно сократить эту запись, если добавить такую инструкцию:

```
using grants::Cary;
```

Теперь имя Cary может использоваться в последующих инструкциях без дополнительного определения. Чтобы иметь возможность ссылаться на обе переменные без определения имени пространства, достаточно написать

```
using namespace grants;
```

 **ПРИМЕЧАНИЕ** В ранних версиях C++ спецификатор namespace не поддерживался.

operator

Спецификатор operator используется для описания функций операторов, переопределяющих встроенные операторы языка (такие как +, -, * и т. д.). Это свойство называется перегрузкой операторов и позволяет переопределить стандартные операторы сложения, вычитания и другие для операций с объектами заданного класса. (См. также главу «Перегрузка операторов».)

Синтаксис:

Для любого заданного оператора @ и комбинации типов соответствующая функция определяется следующим образом:

```
operator@ (type1, type2) // бинарный оператор
operator@ (type1) // унарный оператор
```


private

Модификатор доступа `private` используется для объявления закрытых членов класса. Доступ к таким членам класса могут получить только другие члены этого класса. По умолчанию все члены класса являются закрытыми, поэтому модификатор `private` обычно употребляется в структурах и объединениях (см. также раздел «Структура как разновидность класса» в главе «Классы»).

Синтаксис:

```
private:  
    declarations; // объявления членов класса
```

protected

Модификатор доступа `protected` используется для объявления защищенных членов класса. Доступ к таким членам класса могут получить другие члены этого класса или члены производного класса.

Синтаксис:

```
protected:  
    declarations; // объявления членов класса
```

public

Модификатор доступа `public` используется для объявления открытых членов класса. Доступ к таким членам класса можно получить из любой другой части программы с помощью операторов доступа к членам класса (`.` и `->`).

Синтаксис:

```
public:  
    declarations; // объявления членов класса
```

register

Модификатор `register` применяется для объявления локальных переменных, которые должны размещаться в наиболее быстродействующей памяти (регистровой).

Синтаксис:

```
register data_declaration;
```

Объявленные в качестве аргументов (data_declaration) элементы должны иметь тип указателя или тип целого.

return

Инструкция return возвращает управление из тела функции. При обработке компилятором инструкции return выполнение функции завершается, а все ее последующие инструкции не выполняются.

Синтаксис:

```
return [expression];
```

При задании инструкции return с аргументом expression (который может быть выражением) значение этого аргумента может быть использовано в других функциях. Квадратные скобки показывают, что аргумент expression необязателен. Если аргумент expression опущен, инструкция return просто выполняет выход из функции.

В приведенном ниже примере функция factorial типа long возвращает значение переменной result (n-факториал):

```
long factorial (int n)
{
    // объявление и инициализация переменной:
    long result = 1;
    while (n)
        // вычисление n-факториала
        result = result * n --;
    return result;
}
```

static

Модификатор типа static сообщает компилятору о необходимости статического выделения памяти для переменной или функции.

Синтаксис:

```
static declaration;
```

Здесь `declaration` — объявления переменных.

В приведенном ниже примере выполняется объявление статической локальной переменной. При таком определении переменная существует в течении всего времени выполнения программы. Переменная `number` инициализируется только один раз, при первом вызове функции `func`, и сохраняет свое значение при повторном обращении к функции `func`, в которой она определена:

```
void func (float x)
{
    static int number = 0;
    number ++;
    //...
}
```

В следующем примере выполняется объявление статической функции. Видимость такой функции ограничена ее собственным модулем. По умолчанию функция имеет внешний класс памяти, который позволяет другим модулям в программе обращаться к ней:

```
static void only_my_func (double x);
// прототип функции
//.....
void only_my_func (double x)
{
    statements; // тело функции
};
```



ВНИМАНИЕ Ключевое слово `static` используется только в прототипе функции (то есть при ее объявлении).

В следующем примере осуществляется объявление статических данных — элементов класса. При этом данный элемент становится доступным для всех объектов класса. Чтобы создать статический элемент данных, необходимо не только объявить его в классе, но также определить его

в этом же модуле, как если бы это была глобальная переменная.

Член класса, в котором объект является экземпляром класса, может быть упомянут как `class::member` или как `object.member`. Например:

```
class Study
{
    public:
        static long number;
        Study ( ) { ++ number; }
        //...
};
long Study::number = 0;
```

В конце приведем пример объявления статической функции — элемента класса. Такая функция может обратиться к другому члену класса только в том случае, если он объявлен как статический.

Функция не может использовать ключевое слово `this`, и в отличие от большинства функций-членов класса она не допускает скрытый указатель `this`. Цель статической функции-члена заключается в создании способа обращения к статическим данным-членам:

```
class Study
{
    static long number;
    public:
        Study ( ) { ++number; }
        static long get_number (void);
    // объявление статической функции
    // как элемента класса
    //...
};
long Study::number = 0;
long get_number (void)
{
    return number; // тело статической функции
}
```


struct

Спецификатор `struct` объявляет класс с названием структуры и может содержать элементы разного типа. Единственным отличием такого класса от класса, определенного с помощью спецификатора `class`, является то, что члены класса `struct` по умолчанию открыты (имеют тип доступа `public`). (См. также раздел «union».)

Синтаксис:

```
struct struct_name
{
    type1 member1;
    type2 member2;
    //...
    typeN memberN;
}
object_list;
```

Аргументы:

- `struct_name` — имя типа структуры;
- `typeN memberN` — объявление N-го элемента N-го типа;
- `object_list` — необязательный список переменных типа `struct_name`.

По умолчанию все члены структуры открыты. Для ограничения доступа при описании структуры можно использовать модификаторы доступа `private` и `protected`.

В приведенном ниже примере объявляется структура типа `person`, включающая в себя строковые массивы `surname` и `name`, вложенную структуру типа `data` `birthday` и переменную целого типа `telephone_num`. Данная структура представляет собой набор некоторых сведений о человеке: фамилию, имя, дату рождения, номер телефона:

```
struct data
{
    int day;
    int month;
    int year;
};
```



```
struct person
{
    char surname [15], name [15];
    struct data birthday;
    int telephone_num;
};
```

switch

Инструкция выбора switch реализует условный переход с множественным ветвлением. Инструкция switch сравнивает значения аргумента expression с константами constant1,..., constantN. Когда значение expression равно константе, выполняется следующая за константой последовательность инструкций до тех пор, пока не встретится инструкция break, после которой завершается выполнение всех инструкций блока switch (см. также раздел «case»).

Синтаксис:

```
switch (expression)
{
    case constant1: statement_1list;
        break;
    ...
    case constantN: statement_Nlist;
        break;
    ...
    default:
        statement_default_list;
}
```

Аргументы:

- expression — выражение, приводящееся к целому;
- constantN — целое значение, задающее выполнение последовательности инструкций statement_Nlist.

template

Заголовок template используется для описания шаблонов или параметризованных функций и классов.

Синтаксис:

```
// шаблон класса
template <class arg>
class template_name
{
    declarations;
}
// шаблон функции
template <class arg>
type name (arg)
{
    statements;
}
```

Параметр `arg` — это имя типа параметра, которое компилятор при построении конкретного экземпляра класса заменит указанным типом данных. В этом контексте ключевое слово `class` имеет особое значение и относится к любому типу данных, а не только к объектам, определенным с ключевым словом `class`. Как только шаблон определен, его можно использовать при объявлении объектов, подставляя вместо параметра `arg` определенный тип (`int`, `float`, `double` и т. д.).

В приведенном ниже примере показана процедура создания шаблона класса. Для любого заданного типа `T` (в данном случае это тип `int`) шаблон с именем `pair` определяет новый тип, состоящий из двух членов типа `T` (здесь `int`):

```
template <class T>
class pair
{ // сам шаблон
    T a, b;
};
// применение шаблона для типа int
pair<int> jeans; // переменная jeans
                 // состоит из двух
                 // элементов типа int
```


Определение типа `pair<int>` таким образом превратится в определение класса:

```
class pair<int>
{
    int a,b;
};
```

Пример создания шаблона функции:

```
template <class arg>
type name (args)
{
    <тело функции>
};
```

Следующая функция определяет шаблон для обмена значениями ее аргументов `a` и `b`. Тип аргументов определяется при использовании шаблона:

```
template <class T>
void switch_values (T &a, T &b)
{
    T temp; // объявление переменной temp
    temp = a; // сохранение значения переменной a
    a = b;
    //присвоение переменной a значения b
    b = temp;
    //присвоение переменной b сохраненного значения a
};
```

Далее вызов инструкции

```
switch_values<int>(x,y);
```

приводит к вызову следующей функции:

```
void switch_values (int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
};
```


this

Ключевое слово `this` задает указатель на объект, вызвавший функцию-член класса, поэтому выражение `*this` в теле функции обозначает объект, для которого вызвана эта функция-член.

В следующем модуле ключевое слово `this` используется, чтобы инициализировать переменные `x` и `y` как элементы класса `Point`:

```
Point::set (float x, float y)
{
    this->x = x;
    //инициализирует переменную x класса Point
    this->y = y;
    //инициализирует переменную y класса Point
}
```

В следующем примере с помощью ключевого слова `this` возвращается текущий объект, который используется в функции оператора присваивания. Поскольку `this` — указатель на сам объект, то инструкция `*this` позволяет возвращать этот объект в качестве результата выполнения функции:

```
//Функция оператора присваивания класса MyStr
MyStr& operator=(const MyStr &source)
{
    copy (source.getptr());
    return *this;
}
```

(См. также раздел «Указатель `this` и его использование при создании функций» в главе «Перегрузка операторов».)

throw

Инструкция `throw` применяется для обработки исключений путем их передачи следующему обработчику исключений или генерации нового исключения для его обработки на следующем уровне. (См. также разделы «`try`» и «`catch`».)

Синтаксис:

throw;

или

throw exception_expr;

Здесь `exception_expr` — выражение, которое определяет исключение, вырабатываемое инструкцией `throw`.

Вообще для обработки исключений в C++ используются три инструкции: `try`, `throw` и `catch`. Инструкции, при выполнении которых могут возникать исключительные ситуации и создаваться исключения, помещаются в блок инструкции `try`. При возникновении исключительной ситуации с помощью инструкции `throw` создается исключение, и управление передается первой инструкции `catch`, следующей за блоком инструкции `try` и способной обрабатывать исключения выработанного типа.

Рассмотрим такой пример. Функция `open_file` — функция открытия файла. Если файл не может быть открыт (то есть возникает исключительная ситуация), то управление передается инструкции `throw`, которая вызывает конструкцию `file_err` для создания объекта исключения. Объект исключения типа `file_err` будет затем должным образом обработан в соответствующем блоке инструкции `catch`:

```
void open_file (void)
{
    FILE *fp;
    char *fname = "MY_DOC.TXT";
    fp = fopen (fname, "r");
    if (fp == NULL)
        throw file_err (fname, "r");
    ...
}
```

try

Инструкция `try` используется для обработки исключений. (См. также раздел «`throw`».)

Синтаксис:

```
try
{
    statements;
}
catch (type1 exception)
{
    statements;
}
...
catch (typeN exception)
{
    statements;
}
catch (...)
{
    statements;
}
```

Аргументы:

- `statements` — выражения и инструкции на языке C++;
- `exception` — тип исключения, перехватываемый инструкцией `catch`. Инструкция `catch(...)` позволяет перехватывать исключения всех типов.



ВНИМАНИЕ Каждому исключению должна соответствовать инструкция `catch`, в противном случае при возникновении исключения выполнение программы прекращается.

Рассмотрим пример открытия файла. В этом случае в блоке инструкции `try` выполняются три функции: открытие файла; инициализация переменных; запуск алгоритма. Если при выполнении функции `open_file` произойдет ошибка открытия файла, то управление будет передано на блок инструкции `catch`, после чего выполнение программы завершится. (Описание работы функции `open_file` рассмотрено в примере раздела «throw».)


```
class file_err
{
// объявление элементов класса
public:
    char file [256];
    char mode [10];
    f_error (char *f, char *m)
    {
        strcpy (file, f);
        strcpy (mode, m);
    }
}
try
{
    open_file ( ); // функция открытия файла
    init_vars ( );
//функция инициализации переменных
    run_program ( );
// функция запуска алгоритма
}
catch (file_err f_error)
{
    close_file ( );
    printf ("Ошибка при открытии файла %s
    в режиме %s.", f_error.file, f_error.mode);
    exit (1); //выход из программы
}
```

typedef

Оператор typedef используется для определения новых типов данных.

Синтаксис:

```
typedef declaration;
```

Следующие две инструкции имеют схожее описание, но разные последствия:

```
// объявление типа PTI как указателя
// на переменную типа int *
```



```
typedef int *PTI; ,  
PTI p1, p2, p3; /*  
// последнее эквивалентно int *p1, *p2, *p3;
```



ВНИМАНИЕ Не следует путать оператор определения типа `typedef` с директивой препроцессора `#define`. Директива `#define PTI int*`, если ее использовать в предыдущем примере, приведет к тому, что инструкция, помеченная в этом примере символами `/*`, будет эквивалентна инструкции `int *p1, p2, p3;`. (См. также раздел «`#define`» в главе «Директивы, макросы и операторы препроцессора».)

union

Спецификатор `union` объявляет класс с названием «объединение» и может содержать в одной и той же области данных взаимоисключающие данные разных типов. В отличие от класса, определенного с помощью спецификатора `class`, члены класса `union` по умолчанию открыты (имеют тип доступа `public`). Размер области данных объединения определяется размером члена, занимающего максимальную память. Тип `union` — частный случай типа `struct` (см. раздел «`struct`»).

Синтаксис:

```
union union_name  
{  
    type1 member1;  
    ...  
    typeN memberN;  
} [object_list];
```

Аргументы:

- `union_name` — имя типа объединения;
- `typeN memberN;` — объявление N-го элемента N-го типа;
- `object_list` — список переменных типа `union_name`.

По умолчанию все элементы объединения имеют уровень доступа `public`.

using

Инструкция `using` позволяет упростить доступ к членам пространства имен. (См. также раздел «namespace».)

Синтаксис:

```
using namespace namespace_name;
```

или

```
using namespace_name::member;
```

Аргумент `namespace_name` в первом варианте определяет идентификатор пространства имен, к которому необходим доступ. После этого все члены пространства `namespace_name` становятся видимыми. Во втором случае видимым становится только член `member` пространства `namespace_name`.

virtual

Спецификатор `virtual` используется для объявления виртуальных функций, реализующих свойство полиморфизма родственных классов, или виртуальных базовых классов. Виртуальные функции — это функции-члены, которые объявляются в базовом классе, а затем переопределяются в производных классах. Если функция объявлена виртуальной в базовом классе, она автоматически определяется как виртуальная во всех производных классах, поэтому ключевое слово `virtual` в производных классах применять уже не надо (см. также главу «Виртуальные функции»).

В приведенном ниже примере каждый производный класс `CMom` и `CDad` наследует по одному набору членов виртуального базового класса `CParent`. Класс `CKid` является производным двух базовых классов `CMom` и `CDad`, которые в свою очередь являются производными от `CParent`, несмотря на это, класс `CKid` также наследует только одну копию членов класса `CParent`:

```
class CMom: virtual public CParent
{
//...;
};
```



```
class CDad: virtual public CParent
{
//...;
};
class CKid: public CMom, public CDad
{
//...;
};
```

void

Спецификатор `void` — спецификатор базового типа. Он используется для объявления функций, которые не возвращают никаких значений, а также для объявления универсальных указателей `void*`.

Ниже приведены примеры спецификатора `void`.

Объявление прототипа функции типа `void` с параметром `int n`:

```
void print_num (int n);
```

Объявление прототипа функции без параметров:

```
void get_num (void);
```

Объявление указателя типа `void*`. Такой указатель сохраняет адрес, но не может обратиться к данным. Функции возвращают указатели типа `void*`, когда тип данных неизвестен:

```
void* p = malloc (n);
```

volatile

Модификатор `volatile` используется в объявлении переменной для того, чтобы указать компилятору, что ее значение может изменяться извне, например, системными часами или операционной системой. Таким образом, при каждом обращении программа должна обновлять значение такой переменной путем прямого доступа к памяти. (Требование прямого доступа к памяти особенно важно при

использовании значений, находящихся по адресу, определяемому с помощью указателя.)

Синтаксис:

```
volatile data_declarations;
```

В списке объявлений `data_declarations` могут быть объявлены как переменные, так и указатели. Синтаксис модификатора `volatile` подобен синтаксису ключевого слова `const`, однако только указатель, объявленный как `volatile`, может указывать на данные, объявленные с ключевым словом `volatile`. Тем не менее переменная может быть одновременно объявлена как `const` и как `volatile`. Такое объявление свидетельствует о том, что значение этой переменной может быть изменено системой, но не программой.



ПРИМЕЧАНИЕ Модификатор `volatile` в основном используется в системном программировании.

while

Инструкция `while` используется для создания цикла с предусловием. Так как проверка условия осуществляется перед выполнением тела цикла, возможна ситуация, когда ни одной из инструкций тела цикла выполнено не будет.

Синтаксис:

```
while (condition)
{
    statements;
}
```

Аргументы:

- `condition` — выражение, определяющее условие выполнения тела цикла;
- `statements` — инструкции, выполняемые в теле цикла.

Цикл выполняется до тех пор, пока значение `condition` истинно (TRUE). Если в теле цикла используется только одна инструкция, фигурные скобки можно опустить.

В приведенном ниже примере осуществляется вывод последовательности целых чисел 0, 5, 10, ..., 50:

```
int i;  
while (i <= 50)  
{  
    printf ("%d", i);  
    i += 5;  
}
```


Директивы, макросы и операторы препроцессора

Первый этап, который выполняет компилятор при трансляции программы, называется препроцессорной фазой, то есть фазой предварительной обработки. Во время этого этапа считываются файлы заголовков, выбираются строки программы, которые требуется компилировать, и выполняется текстовая замена. Работу компилятора на этом этапе называют препроцессорной обработкой.

Преимуществом препроцессорной обработки является то, что она выполняется перед компиляцией и выполнением программы, не увеличивая само время выполнения.

Ниже приведены три типа элементов, обрабатываемых препроцессором:

- директивы, которые выполняют специальные команды перед компиляцией программы;
- предопределенные (определенные в C++) макросы, вместо имен которых в тексте программы на этапе компиляции будут вставлены выполняемые этими макросами действия;
- операторы препроцессора, которые используются в директивах `#if` и `#define`.

Синтаксис директив препроцессора отличается от синтаксиса инструкций C++:

- директива заканчивается в конце строки, а не точкой с запятой (;), в качестве символа продолжения строки используют обратный слэш (\);
- директивы должны начинаться символом # в первой позиции строки;
- пробел в директиве является разделителем.



ВНИМАНИЕ В этом разделе широко используется понятие символической константы, являющейся просто идентификатором, вместо которого препроцессор подставит реальные фрагменты программы (это могут быть константы C++, выражения, инструкции), после этого выполняется компиляция полученного текста. Не путайте *символические* и *символьные* константы. Последние располагаются непосредственно в памяти программы (см. также главу «Словарь терминов и понятий»).

Директивы

Часть директив обеспечивает поддержку условной компиляции, используемой, например, для получения нескольких версий одной программы. К ним относятся следующие шесть директив: `#if`, `#ifndef`, `#elif`, `#ifdef`, `#else`, `#end`.

Остальные директивы предоставляют возможность включения файлов заголовков и макроопределителей: `#define`, `#include`, `#pragma`, `#error`, `#line`, `#undef`.

Далее в алфавитном порядке детально описывается каждая директива.

#define

Эта директива применяется в трех случаях.

- Для создания символических констант.

Синтаксис: `#define identifier replacement` (вместо идентификатора `identifier` во всем файле подставляется фрагмент `replacement`, который может быть константой, выражением или инструкцией).

- Для записи макрофункций, которые выглядят как вызов функции, но выполняются путем замены имени такой функции ее определением без использования обычного механизма вызова функций (см. также раздел «Функции» в главе «Основные особенности программирования на C++»).

Синтаксис: `#define identifier(arg1 [, arg2]...) replacement` (вместо идентификатора `identifier` во всем файле подставляется фрагмент `replacement`, причем при вызове

макроса в тексте replacement параметры arg1, arg2 и т. д. заменяются их значениями).

- Для определения идентификатора, что обеспечивает управление процессом компиляции.

Синтаксис: #define identifier (вместо идентификатора identifier во всем файле подставляется пробел, и идентификатор identifier считается определенным):

Квадратные скобки ([]) и многоточие (...) означают, что количество параметров, разделенных запятыми, не ограничено. Первый появившийся пробел (не считая пробелов в круглых скобках или в кавычках) завершает подставляемый текст replacement.

Напомним, что для создания длинных строк подставляемого текста можно использовать символ продолжения строки (\).

Символические константы

При работе с числами, особенно если они состоят из большого количества цифр, для сокращения их записи полезно использовать символическую запись (эта запись также называется символической константой). Как правило, в отличие от имен переменных, символические константы обозначаются заглавными буквами.

Например:

```
#define PI 3.15159265
```

Теперь, где бы ни появился идентификатор PI, препроцессор заменит его значением 3.14159265.



ВНИМАНИЕ Идентификатор не заменяется, если он является частью комментария, строки символов, заключенной в кавычки, или более длинного идентификатора (например, фрагмент PI в составе идентификатора PIG не заменяется).

Тот же результат можно получить, определяя PI как константную переменную. Например:

```
const double pi = 3.14159265;
```


Различие при использовании этих двух методов заключается в следующем. Если константа PI определена как символическая, то ее использование равнозначно прямой записи числа в тексте программы. Если выражение состоит из таких символических констант и чисел, указанных в явном виде, то оно вычисляется на этапе компиляции, вместо выражения подставляется его значение и время выполнения программы сокращается. Например, в следующем примере выражение $\text{PI}/2$ будет вычисляться уже во время компиляции, а не во время выполнения программы:

```
(x = sin(y * (PI/2)));
```

Символические константы также часто используются для обозначения размеров массивов, что до минимума сокращает количество изменений программы при изменении размера массива — достаточно просто изменить значение соответствующей символической константы в директиве `#define`:

```
#define ROWSIZE 30
#define COLSIZE 20
//...
int matrix1[ROWSIZE][COLSIZE];
int matrix2[ROWSIZE][COLSIZE];
```

В этом примере размер матриц определен как 20×30 . Для матриц размером 40×30 достаточно изменить первые две строки:

```
#define ROWSIZE 30
#define COLSIZE 40
```

С другой стороны, константные переменные (объявленные как `const`) полезны при компиляции ошибок и отладке, так как по именам переменных можно легко узнать их значения (подробнее об этой возможности рассказывается в описании компилятора).



ПРИМЕЧАНИЕ Переменные могут быть объявлены константными и для того, чтобы область их видимости ограничивалась классом. При этом поддерживается основной принцип объектно-ориентированного программирования.

Таким образом, как символические константы, так и переменные, объявленные с модификатором `const`, имеют свои преимущества и недостатки.

Макрофункции

Макрофункции (их также называют макросами) выглядят как вызов функции, но выполняются не через реальный механизм вызова функции, а путем подстановки тела макрофункции вместо ее имени.

Если дано следующее определение макроса, то перед компиляцией препроцессор выполнит замену всех вызовов этого макроса в тексте программы:

```
#define max(A/B) ((A)>(B)?(A):B))
```

Например, вызов `max(i, j)` будет заменен следующим:

```
((i)>(j)?(i): j))
```

Макрофункции применяются для определений небольших функций, так как они ускоряют выполнение программы, но увеличивают используемую программой область памяти. К другим недостаткам макрофункций относится невозможность определения типов параметров. Решение проблемы обеспечивают встраиваемые функции (см. также раздел «Определение встраиваемой функции» в главе «Классы»). Как и у макрофункций, тело встраиваемой функции непосредственно подставляется в программу, что увеличивает скорость ее выполнения.



ВНИМАНИЕ Заключение в круглые скобки каждого параметра в части `replacement` заставляет компилятор подставлять имена параметров в тело функции до ее выполнения. Если круглые скобки не указывать, результат будет непредсказуемым.

Управление компиляцией

Определение идентификатора используется в C++ для директив `#ifdef` и `#ifndef`. Директивам `#ifdef` и `#ifndef` не важно, каким является значение идентификатора, важно, что он просто определен. В такой ситуации можно опреде-

лять идентификатор без каких-либо значений (по умолчанию он заменяется пробелом).

В следующем примере идентификатор `USE_8086` определен, что заставляет препроцессор компилировать код между директивами `#ifdef` и `#endif`:

```
#define USE_8086
```

```
//...
```

```
#ifdef USE_8086
```

```
//Оптимизация для процессора 8086...
```

```
#endif
```

#elif

Директива `#elif` (else if) вставляет альтернативное условие как продолжение блока `#if...#endif`. Эта директива определяет, выполняется ли условие компиляции таким же образом, как директива `#if`.

Синтаксис:

```
#elif constant_expression
```

В выражение `constant_expression`, так же как в директиве `#if`, могут входить любые константы, кроме оператора `sizeof`, операторов приведения типа и перечисляемых типов, объявленных с помощью ключевого слова `enum`. Выражение может включать в себя специальный оператор `define(symbol)`, где `symbol` — символическое имя (см. также раздел «Операторы препроцессора»). Этот оператор препроцессора возвращает единицу, если имя `symbol` было определено, и ноль, если нет (см. также раздел «`#if`»).

#else

Директива `#else` отмечает начало последнего блока в последовательности директив условной компиляции и компилирует его в том случае, если ни одно из предыдущих условий `#if` и `#elif` не выполняется (см. также раздел «`#if`»).

По смыслу эта директива совпадает с инструкцией `else`, но используется для проверки условия во время компиляции, а не во время выполнения программы.

Блок `#else` завершается директивой `#endif`:

```
//...  
#else  
    statement_block_n+1  
#endif
```

#endif

При использовании `#if` директивы или родственных ей директив `#ifdef` и `#ifndef` конец блока условного компилирования обозначается отдельной строкой `#endif`:

```
#endif
```

(См. также раздел «`#if`».)

#error

Директива `#error` останавливает компиляцию и выводит на экран сообщение об ошибке:

```
#error сообщение
```

Например, следующий код останавливает компиляцию и выводит на экран сообщение, если символ `ERR` определен:

```
#ifdef ERR  
#error Error is here!  
#endif
```

Когда компилятор выполняет директиву `#error`, он останавливается, печатает номер текущей строки и сообщение `Error is here!`.

Как правило, директива используется, если дальнейшая компиляция программы не имеет смысла.



СОВЕТ

Правильно выбранное сообщение поможет в дальнейшей отладке.

#if

Внешне, да и по смыслу, директива `#if` очень схожа с условной инструкцией `if` (см. раздел «Управляющие инструкции» в главе «Основные особенности программирования на C++»). Но директива `#if` проверяет условие во время компиляции, а инструкция `if` — во время выполнения программы. Существует два основных случая использования директивы `#if`:

- условная компиляция. Особенно полезно ее применение для компиляции различных вариантов одной программы, не дублируя текст этой программы;
- комментирование фрагментов программы.

Директива `#if` обозначает начало условного блока компиляции. Если выражение, обозначенное директивой `#if`, ненулевое (`true`), то процессор C++ компилирует следующую часть программы до тех пор, пока не встретит одну из директив `#elif`, `#else`, `#endif`.

Синтаксис:

```
#if constant_expression
statement_block_1
[ #elif constant_expression
statement_block_2 ]
[ #elif constant_expression
statement_block_3 ]
...
[#else
statement_block_n+1]
#endif
```

Квадратные скобки обозначают необязательные пункты: необходимы лишь директивы `#if` и `#endif`. Можно использовать любое количество директив `#elif`, директива `#else` записывается после всех директив `#if` и `#elif`. После каждой директивы `#if` может быть записана только одна директива `#else`.

Препроцессор последовательно проверяет каждое выражение `constant_expression` до тех пор, пока не найдет ненуле-

вое (имеющее значение true). В этом случае компилируется соответствующий блок инструкций `statement_block`. Блок `statement_block` может содержать любой фрагмент программы C++, включая описания, директивы и инструкции.

Каждое выражение `constant_expression` может состоять из констант и операторов, но не может содержать оператор `sizeof`, не может выполнять приведение типов и использовать перечисляемые типы, объявленные с помощью ключевого слова `enum`. Эти выражения часто содержат символы, ранее определенные директивой `#define`, или параметры компиляции.

Условная компиляция

Условная компиляция — это удобный путь для создания нескольких версий одной программы, например, при необходимости написания программы для разных платформ и определения той части программы, которая будет реализована в данном случае; для построения отладочной версии, которая выдает периодические оценочные сообщения, а также окончательной версии.

Тут возникает дилемма. Постоянно переключаясь с версии на версию, проверяя их, приходится совершать много лишней работы, добавляя и удаляя программный код. При этом хранение нескольких полных версий программ занимает много места на диске. А главное, каждый раз при добавлении новой команды придется просматривать и исправлять все версии.

Решением последней проблемы является использование условной компиляции, которая позволяет выделять части кода, различающиеся в каждой версии.

Например, пусть программа использует различные системы координат для различных типов компьютеров. Введем следующие обозначения констант:

```
#define SHORT 0
#define LONG 1
#define REAL 2
```


Затем для выбора нужной системы координат введем символ COORDSYS (как обычно, используем заглавные буквы для обозначения символической константы). Теперь, для того чтобы определить нужный вариант компиляции, достаточно написать одну строку:

```
#define COORDSYS REAL
```

Многие компиляторы поддерживают параметры командной строки (строки, задающей параметры компиляции и запускающей эту компиляцию) или ключи (параметры) среды программирования, что позволяет определить параметры компиляции без использования директивы `#define`. Названия таких параметров можно узнать в документации по компилятору:

```
#if COORDSYS == SHORT
    short x, y;
#elif COORDSYS == LONG
    long x, y;
#elif COORDSYS == REAL
    double x, y;
#endif
```

Комментарии

Иногда требуется удалить некоторый фрагмент программы так, чтобы потом можно было легко вернуть его обратно. Самый очевидный путь — отметить начало и конец удаляемого фрагмента с помощью символов комментария `(/*)` и конца `(*/)` соответственно. Но этого иногда недостаточно, например, если внутри скрываемого блока уже находятся комментарии.

Чтобы временно приостановить выполнение программного кода, можно использовать директивы `#if 0 ... #endif`. В этом случае при выполнении условия компилятор проигнорирует выделенные строки:

```
#if 0
    int i, j; /* Объявление i и j как int */
    double x, y; /* Объявление x и y как double */
#endif
```


#ifdef

Директива `#ifdef` является наиболее часто используемой из всех условных директив (`#if`, `#ifdef`, `#ifndef`).

Синтаксис:

```
#ifdef symbol
```

или

```
#if defined(symbol)
```

Здесь `symbol` — символическое имя. Если аргумент `symbol` определен, то строки, следующие за директивой `#ifdef`, компилируются до тех пор, пока не появится одна из следующих директив: `#elif`, `#else` или `#endif`. Само значение идентификатора `symbol` не важно, важно, чтобы он просто был определен. Например, значением идентификатора `symbol` может быть пробел, как в следующем определении идентификатора `32_BIT_SUPPORT`:

```
#define 32_BIT_SUPPORT
```

Многие компиляторы обладают ключами для определения таких символов, которые позволяют управлять компиляцией из командной строки, командного файла (`batch`-файла), файла для построения задачи (`make`-файла) или среды программирования.

(См. также раздел «`#if`».)

#ifndef

Директива `#ifndef` используется для условной компиляции.

Синтаксис:

```
#ifndef symbol
```

или

```
#if ! defined(symbol)
```

Здесь `symbol` — символическое имя.

Если идентификатор не определен, то компилируются следующие за `#ifndef` строки до появления одной из дирек-

тив: `#elif`, `#else` или `#endif`. Директивы `#ifndef` и `#ifdef` особенно полезны, когда нужно компилировать фрагмент программы в зависимости от простой переменной, принимающей два значения: ноль (0, false) и единица (1, true).

В следующем примере выполняется проверка включения в программу файла заголовков `myproj.h` и задается соответствующее условие. Если этот файл еще не был включен в программу, он включается. Показанный ниже фрагмент можно вставлять в любое количество различных файлов, любое количество раз, но файл `myproj.h` будет включен только один раз:

```
#ifndef HEADER_INCLUDED
#include "myproj.h"
#define HEADER_INCLUDED
#endif
```

(См. также раздел «`#if`».)

#include

Директива `#include` заставляет препроцессор C/C++ включать в файл программы другой файл. Как правило, таким образом в программу включаются файлы заголовков, содержащие прототипы используемых функций, в том числе библиотечных. Например, при задании директивы `#include<stdio.h>` в текст программы будет добавлено все содержимое файла `stdio.h` (это, однако, не приведет к изменению исходного файла).

Синтаксис:

```
#include "filename" //первая строка программы
```

или

```
#include <filename> //первая строка программы
```

Все файлы, включенные в программу с помощью директивы `#include`, компилируются так, как будто их содержимое является частью файла программы. Включенный файл, в свою очередь, также может содержать директивы `#include`, включающие в него другие файлы заголовков, и т. д. Глуби-

на вложенности таких файлов зависит только от размеров свободной памяти компьютера.

Оба варианта записи директивы выполняют поиск указанного файла в стандартной папке для файлов заголовков (ее имя указывается в параметрах компиляции). Однако при первой форме записи сначала поиск выполняется в текущей папке (папке, в которой расположен файл проекта).

Для определения файла заголовков, который был создан самим программистом и обычно хранится в текущей рабочей папке, нужно использовать директиву `#include "filename"`. Второй синтаксис предпочтительнее для стандартных файлов заголовков, таких как `stdio.h` и `stdlib.h`. Например:

```
#include <stdio.h>
```

 **ПРИМЕЧАНИЕ** Согласно ANSI C++ для включения прототипов стандартных библиотечных функций из файла заголовков `stdio.h` можно использовать как директиву `#include<stdio.h>`, так и директиву `#include<cstdio>`. Последний вариант включает заголовки функций ввода-вывода прямо из виртуального файла заголовков и может не приводить к реальному включению текста файла заголовков в файл программы.

#line

Директива `#line` вставляет новый номер строки и новое имя исходного файла, что отражается макросами `__LINE__` и `__FILE__`. Эта директива используется очень редко и не оказывает никакого действия на физический исходный файл.

Синтаксис:

```
#line num ["filename"]
```

После выполнения компилятором этой строки текущему номеру следующей строки присваивается номер `num`, нумерация следующих строк продолжается с нового номера. Если файл `filename` определен, он становится новым значением переменной `__FILE__`, после чего макрос `__FILE__`

сохраняет кавычки как признак строки символов. Например:

```
#line 100 "source1"
#include <stdio.h>
//...
void main()
{
    print("%d: %s \n", __LINE__, __FILE__);
    //...
}
```

Результатом выполнения этого кода станет напечатанная строка: 103: source1.

#pragma

Директива `#pragma` позволяет использовать специфичные для конкретных реализаций директивы.

Синтаксис:

```
#pragma arguments
```

Аргумент `arguments` содержит имя директивы.

При помощи этой директивы C++ позволяет определить любые желаемые директивы, не обращаясь к другим, поддерживающим их компиляторам. Если компилятор не поддерживает имя директивы, указанное в параметре `arguments`, он просто игнорирует директиву `#pragma`, не выдавая при этом никаких сообщений об ошибках или предупреждений.

С помощью директивы `#pragma` можно, например, задать программе функцию (или несколько функций), которые должны вызываться либо при загрузке перед вызовом функции `main`, либо непосредственно перед выходом из программы. Эти директивы имеют следующий синтаксис:

```
#pragma startup func_name
#pragma exit func_name
```

В этом случае аргумент `func_name` содержит имя ранее объявленной функции, не принимающей аргументов и возвра-

щающей значение `void` (то есть эта функция должна быть объявлена как `void func_name (void)`).

(См. также документацию компилятора.)

#undef

Директива `#undef` позволяет удалить определение символического имени (ранее оно может быть определено с помощью `#define` или в ключах компиляции). Эта директива полезна в том случае, когда нужно компилировать фрагмент программы в зависимости от простой переменной, принимающей два значения: ноль (0, false) и единица (1, true).

Синтаксис:

```
#undef symbolname
```

Здесь `symbolname` — некоторое символическое имя.

Когда препроцессор обрабатывает эту строку, он переводит `symbolname` в неопределенное состояние. Например, следующий код временно определяет идентификатор `LINESZ`, а в конце данного примера значение идентификатора `LINESZ` становится неопределенным:

```
#define LINESZ 81
//...
char inputstr[LINESZ];
//...
#undef LINESZ
```



ПРИМЕЧАНИЕ Директива `#undef symbolname` будет выполняться даже в том случае, если идентификатор `symbolname` не был определен.

Предопределенные макросы

Язык C++ определяет набор символических имен, которые служат для определения состояния дел во время компиляции: какая программа запускается там, где появилась ошибка и т. д.

Ниже приведены предопределенные символические имена, их также называют макросами:

- `__cplusplus` — определен, если начальный файл был скомпилирован как файл C++;
- `__DATE__` — строка, заключенная в кавычки и содержащая дату компиляции;
- `__FILE__` — строка, заключенная в кавычки и содержащая имя исходного файла;
- `__LINE__` — номер текущей строки;
- `__TIME__` — строка, заключенная в кавычки и содержащая время компиляции.

Рассмотрим эти макросы более подробно.

`__cplusplus`

Макрос `__cplusplus` определен, если исходный файл был скомпилирован как файл C++. Большинство компиляторов C++ могут считывать файлы, имеющие расширение C (*.c), компилируя их как инструкции языка C. Этот макрос также задает возможность использования функций C++. Например:

```
#ifdef __cplusplus
//используется особенность C++
int i = get_random();
#endif
```

Если исходный файл был скомпилирован как код языка C, то возникнет ошибка, так как происходит вызов функции при объявлении переменной. Использование директивы `#ifdef` и макроса `__cplusplus` позволяет предотвратить эту ошибку. Такая возможность полезна при многочисленном копировании кодов C/C++ друг в друга. К счастью, подобные ситуации встречаются не часто.

(См. также приложение «Отличия C и C++».)

`__DATE__`

Во время компиляции макрос `__DATE__` преобразуется в строку, ограниченную кавычками, которая содержит теку-

щую дату компиляции, в виде: *месяц день год*. Следующий пример позволяет выдать сообщение: «Дата компиляции Feb 22 2000», при условии, что текущая дата 22 февраля 2000 года:

```
#include <stdio.h>
//...
printf("Дата компиляции %s.\n", __DATE__);
```

__FILE__

Макрос `__FILE__` транслируется в строку, заключенную в кавычки, которая содержит имя текущего исходного файла. Например:

```
#include <stdio.h>
//...
printf("Ошибка в файле %s.\n", __FILE__);
```

Если исходный файл содержится в папке `c:\samples\test.cpp`, то макрос учитывает имя папки и выдает полное имя файла в виде символьной строки.

В результате выполнения примера будет напечатана строка:
Ошибка в файле `c:\samples\test.cpp`.

__LINE__

При компиляции макрос `__LINE__` транслируется в номер текущей строки компилируемого исходного файла и может изменяться директивой `#line`:

```
#include <stdio.h>
//...
printf("Ошибка в строке %d.\n", __LINE__);
```

Если текущая строка является семнадцатой, то макрос транслируется в строку, заключенную в кавычки и содержащую этот номер.

В результате выполнения примера будет напечатана следующая строка:

Ошибка в строке 17.

__TIME__

Макрос `__TIME__` компилируется в строку, заключенную в кавычки и содержащую текущее время компиляции в формате: часы:минуты:секунды. Например:

```
#include <stdio.h>
//...
printf("Время компиляции %s.\n", __TIME__);
```

Если время по часам компьютера при компиляции последней строки примера было 10 часов 5 минут 30 секунд, то в результате выполнения примера будет выведена строка:

Время компиляции 10:05:30.

Операторы препроцессора

Существует несколько операторов, которые работают только в течение препроцессорной фазы компиляции:

- `defined (symbol)` — возвращает единицу (1), если символ определен, и ноль (0) в противном случае;
- `#str` — превращает строку `str` в выделенную строку (то есть строку, заключенную в кавычки);
- `text1##text2` — конкатенирует (объединяет) текстовые фрагменты `text1` и `text2`.

Здесь `symbol`, `str`, `text1` и `text2` — символические имена. Последние два оператора (`#` и `##`) используются в поле перестановки (`replacement`) директивы `#define`. Оператор `#` ставит аргумент в кавычки, например, при компиляции препроцессором строк:

```
#define pr(s) puts(#s)
pr(Hi there);
```

Результат будет представлен в виде:

```
puts("Hi there");
```

Оператор `##` объединяет два текстовых фрагмента в один. Например:

```
#define call(verb, adj, do) verb##_##adj(do)
```


Допустим, в программе макрофункция `call` вызывается со следующими параметрами:

```
call(shrink, all, trees);
```

Тогда компилятор переводит этот вызов в вызов функции:

```
shrink_all(trees);
```

Операторы `#` и `##` редко используются в программах. Тем не менее они увеличивают возможности работы с текстом во время препроцессорного этапа компиляции.

Вот пример, имеющий отношение к практике. Библиотека имеет несколько версий одной функции:

```
strcpy  
wstrcpy
```

Проблема в том, что, записывая определенные макросы, нужно выбирать одну из этих функций уже во время компиляции. Самый эффективный путь — создать имя с помощью текстовых операторов. Вот примерная запись создания имени функции:

```
#define STRINGF PREFIX##str##OP
```

Зададим следующие определения:

```
#define PREFIX w  
#define OP cpy
```

Тогда макрос `STRINGF` будет транслироваться в следующее имя функции:

```
wstrcpy,
```

что позволяет другой строке программы определить, используется ли `w`-версия функции (для широкого символьного формата) в коде, когда появляется макрос `STRINGF`. Значение `PREFIX` может быть введено в командной строке компилятора, так что никогда не понадобится переводить строки кода из формата в формат.

Как и остальные операторы препроцессора, оператор `##` используется редко.

Библиотечные функции

Спецификация ANSI языков C и C++ определяет богатый набор функций стандартной библиотеки. Среди прочего в них определяются функции, позволяющие читать и писать файлы, вводить данные с клавиатуры и выводить их на монитор, получать значение времени суток и производить математические вычисления.

Хотя, строго говоря, библиотека и сами языки — совершенно разные вещи, любой поставщик компиляторов реализует, как минимум, стандартную библиотеку. Во многих случаях реализуется гораздо больший набор функций, чем описано здесь; так, зачастую библиотека расширяется функциями работы со специальными типами данных и функциями поддержки различных элементов операционной системы и аппаратной платформы. Расширенный набор функций, однако, лишает возможности писать переносимые программы, поэтому без особой необходимости не рекомендуется использовать функции, не включенные в данную главу.

Стандартная библиотека языка C++ совпадает со стандартной библиотекой языка C. Возможности, доступные в C++, но отсутствующие в C, реализуются посредством классов и не входят в стандартную библиотеку, чем обеспечивается их совместимость. Упомянутые классы описываются в главе «Объекты и классы библиотеки ввода/вывода».

Обзор библиотечных функций

Несмотря на то что стандартная библиотека очень велика, в ней можно выделить несколько больших функциональных групп.

- Стандартные функции ввода/вывода. Их описания включены в файл заголовков `stdio.h` и виртуальный заголовок `cstdio`. В эту группу включают функции кон-

сольного ввода/вывода, такие как `printf`, `scanf`, `gets`, `puts`, `getchar` и `putchar`. Сюда также входят функции файлового ввода/вывода `fopen` (для открытия файлов), `fprintf`, `fscanf`, `fputs`, `fgets` и ряд других (для чтения/записи текстовых файлов), `fread` и `fwrite` (для чтения/записи двоичных файлов). Также к этой группе относятся функции получения текущей позиции в файле и установки новой позиции в файле (`fseek`, `ftell`, `rewind`), которые позволяют организовать произвольный доступ к данным в файле. (См. разделы «Функции консольного ввода/вывода» и «Функции файлового ввода/вывода».)

- Функции работы со строками. Их описания включены в файл заголовков `string.h` и виртуальный заголовок `cstring`. В эту группу входят функции работы с простыми строками типа `char*`: `strlen` возвращает длину строки, `strcpy` копирует строку в другую и `strcat` добавляет строку в конец строки. Функция `strcmp` используется для сравнения двух строк на совпадение. Другие важные функции: `strncat`, `strncmp`, `strncat` и `strstr`. Есть также ряд функций типа `strchr` для поиска требуемого символа. Функция `strtok` полезна для разбивки строки на отдельные элементы. (См. раздел «Функции работы со строками».)
- Функции определения типа символа. Их описания включены в файл заголовков `ctype.h` и виртуальный заголовок `sctype`. Эти функции позволяют определить принадлежность символа к определенной категории. Например, `isalpha` возвращает истину при условии, что анализируемый символ является буквой английского языка. Преимущество этих функций заключается в том, что они позволяют писать переносимые программы, не зависящие от используемой кодировки символов (ASCII, EBCDIC и т. п.). В эту группу функций также входят: `isalnum`, `isctrl`, `isdigit`, `isgraph`, `islower`, `ispunct`, `isspace`, `isupper` и `isxdigit`. (См. раздел «Функции проверки категории символов».)

- Математические функции. Их описания включены в файл заголовков `math.h` и в виртуальный заголовок `cmath`. К этой группе относятся такие стандартные тригонометрические и гиперболические функции, как `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`, `sinh`, `cosh` и `tanh`. Сюда же включаются функции `pow` для вычисления степени числа, `exp` для вычисления экспоненциальной функции и логарифмические функции `log` и `log10`, а также функция для вычисления ближайшего большего целого `ceil`, функции для вычисления ближайшего меньшего целого `floor`, `fabs`, `fmod`, `frexp`, `ldexp`, `modf` и функция вычисления квадратного корня `sqrt`. Необходимо отметить, что такие математические по своей природе функции, как `abs` и `rand`, описаны в файле `stdlib.h`, а не в файле `math.h`.
- Функции работы с датой и временем. Их описания включены в файл заголовков `time.h` и в виртуальный заголовок `ctime`. В эту группу включается функция `time`, которая возвращает текущую дату и время в структуре `time_t`. Большинство других функций этой группы предназначены для перевода значения типа `time_t` в определенный формат, это функции `asctime`, `ctime`, `difftime`, `gmtime`, `localtime`, `mktime` и `strftime`. К этой группе относятся и функции, позволяющие определить время работы программы. (См. раздел «Функции работы с датой и временем».)
- Функции работы с памятью. Их описания включены в файл заголовков `stdlib.h` и в виртуальный заголовок `cstdlib`. В основном эти функции используются в библиотеке для совместимости с языком C. К ним относятся функции `malloc`, `free`, `calloc` и `realloc`. В языке C++ для тех же целей преимущественно используются операторы `new` и `delete`.
- Функции с переменным списком параметров. Их описания включены в файл заголовков `stdarg.h`. Эти функции (реально они являются не функциями, а макросами) позволяют обрабатывать список параметров переменной

длины. Примером использования этих функций является функция `printf`. В эту группу включаются также функции `va_start`, `va_arg` и `va_end`. (См. раздел «Поддержка списков параметров переменной длины».)

- Прочие функции. Ряд функций трудно отнести к какой-либо группе. Описания большинства из них (но не всех) включены в файл заголовков `stdlib.h` и в виртуальный заголовок `cstdlib`. К этой группе относятся некоторые математические функции, такие как `abs`, `labs` и `ldiv` и `srand` (для генерации псевдослучайных чисел), функции взаимодействия с операционной системой `system` и `getenv`, функции завершения процесса `abort`, `exit` и `atexit`, функции сортировки и поиска в больших массивах `qsort` и `bsearch`. Последним набором функций, которые включены в файл заголовков `stdlib.h`, являются функции, оставленные в C++ для обратной совместимости с языком C. К ним относятся `setjmp`, `longjmp`, `raise` и `signal`.

Пункты, в которых представлены описания функций стандартной библиотеки, построены по следующей схеме:

- прототип функции с описанием всех ее параметров, их типов и типа возвращаемого функцией значения;
- имя файла заголовка, которое обязательно должно включаться в файл программы с помощью директивы `#include`;
- описание выполняемых функцией действий;
- имена функций, используемых вместе с данной функцией или близких по выполняемым действиям.

Поддержка списков параметров переменной длины

Прототипы:

```
void va_start(va_list argptr, param);  
type va_arg(va_list argptr, type);  
void va_end(va_list argptr);
```

Файл заголовков: `stdarg.h`.

В стандартной библиотеке C/C++ определен ряд макросов для обработки списков аргументов переменной длины. Функция, в которой используется такой механизм, должна иметь хотя бы один аргумент: макрос `va_start` использует его в качестве «якоря», при помощи которого можно добраться до остальных аргументов (примером такой функции может служить функция `printf`). Макросы приведены в табл. 7.1.

Таблица 7. Макросы

Макрос	Действие
<code>va_start(argptr, param)</code>	Инициализация списка аргументов на основе <code>argptr</code> (переменная типа <code>va_list</code> — см. ниже), «якорем» служит последняя фиксированная в прототипе переменная, ее имя передается в качестве второго аргумента макроса <code>param</code>
<code>va_arg(argptr, type)</code>	Возвращает следующий аргумент из списка, причем <code>type</code> — тип аргумента
<code>va_end(argptr)</code>	Указывает, что работа со списком закончена

Для получения доступа к переменному списку аргументов необходимо сначала инициализировать служебную переменную `argptr` типа `va_list` (также определенную в файле `stdarg.h`). После ее определения появляется доступ к списку аргументов функции:

```
va_list argptr;
```

В следующем примере показана функция, имеющая переменный список аргументов, имеющих тип `int`. Эта функция печатает каждое значение, а затем печатает полную сумму этих чисел. Единственный фиксированный аргумент `num_of_args` содержит число последующих аргументов:


```
#include <stdarg.h>
#include <stdio.h>
void print_ints(int num_of_args,...)
{
    int total, i;
    va_list ap;
    va_start(ap, num_of_args);
    while (num_of_args--)
    {
        i = va_arg(ap, int);
        printf("%d\n", i);
        total += i;
    }
    printf("Total summ is %d\n", total);
    va_end(ap);
}
```

Рассмотрим следующий вызов функции:

```
print_ints(3, 20, 35, 15);
```

В этом случае вывод будет таким:

```
20
35
15
Total summ is 70
```

А если параметры будут следующими:

```
print_ints(4, 19, 34, 14, 4);,
```

вывод будет выглядеть так:

```
19
34
14
4
Total summ is 71
```

Функции консольного ввода/вывода

Прототипы:

```
int getchar(void);
char *gets(char *s);
```



```
void perror(const char *s);  
int printf(const char *fmt, ...);  
int putchar(int ch);  
int puts(char *s);  
int scanf(const char *fmt, ...);
```

Файл заголовков: `stdio.h` или `cstdio`.

Стандартная библиотека C/C++ включает ряд функций для чтения и записи на консоли (клавиатура и монитор). Эти функции читают и пишут данные как простой поток символов (см. раздел «Понятие потока» в главе «Ввод и вывод»).

(См. также раздел «Функции файлового ввода/вывода».)

Обзор

Ниже приведены основные функции консольного ввода/вывода:

- `getchar()` возвращает очередной символ с клавиатуры как целое;
- `gets(s)` читает символы с клавиатуры до появления символа новой строки и помещает их в строку `s` (сам символ новой строки в строку не включается). Возвращает указатель на `s`;
- `perror(s)` печатает строку вида «`s: error сообщение`», в которой сообщение определяется на основе значения глобальной переменной `errno`. Если значением `s` является `NULL`, то печатается только сообщение об ошибке. Вывод производится в стандартный поток ошибок `stderr`, обычно назначенный на монитор;
- `printf (fmt, par1, par2, ...)` выводит строку параметров `par1`, `par2` и т. д. в формате, определенном строкой `fmt`, на стандартное устройство вывода (обычно монитор). Возвращает число выведенных символов (см. описание функции `printf`);
- `putchar(ch)` выводит символ `ch` на стандартное устройство вывода. Если вывод успешен, возвращается значение `ch`, иначе возвращается значение `E0F`;

- `puts(s)` выводит строку `s` на стандартное устройство вывода, добавляя в конце символ новой строки. Возвращает неотрицательное значение при успехе или EOF — при ошибке;
- `scanf (fmt, par1, par2, ...)` вводит строку параметров `par1, par2` и т. д. в формате, определенном строкой `fmt`, со стандартного устройства ввода (обычно с клавиатуры). Возвращает число переменных, которым присвоено значение (см. описание функции `scanf`).

Чтение символов с клавиатуры

Чтение одиночных символов производится функцией `getchar`. Операционная система зачастую реализует ввод с клавиатуры через буфер, то есть `getchar` не сможет что-либо вернуть до тех пор, пока на клавиатуре не нажата клавиша Enter:

```
int c;  
c = getchar(); // получим следующий символ
```

В стандартной библиотеке нет средств для получения отдельных символов, не дожидаясь символа конца строки. Тем не менее в конкретных версиях C++ такая функция обычно предусматривается (это может быть, например, `getch()`) (см. также документацию компилятора).

Функция `gets` получает всю строку до символа конца строки и возвращает ее в буфер, выделенный для нее в программе. Функция не проверяет длину строки, поэтому в программе должен использоваться буфер достаточной длины, чтобы не произошло его переполнение:

```
char s[81];  
gets(s);
```

Необходимо помнить, что функция `gets` не заносит символа новой строки в полученную строку.

Альтернативой функции `gets` является функция `fgets` с указанием `stdin` в качестве входного потока. В отличие от `gets` функция `fgets` ограничивает число помещаемых в строку символов, а также не удаляет символ новой строки.

Чтение чисел с клавиатуры

Функция `scanf` представляет собой средство ввода чисел с клавиатуры (см. описание функции `scanf`):

```
int i;  
scanf("%d", &i);
```



ВНИМАНИЕ В списке параметров функции `scanf` должны указываться адреса вводимых переменных, а не сами переменные (см. описание функции `scanf`).

Несмотря на то что `scanf` прекрасно работает в простых программах, для больших программ она оказывается недостаточно гибкой. Функция `scanf` не завершается до тех пор, пока пользователь не введет число и не нажмет клавишу Enter. Если клавиша Enter будет нажата без предварительно введенного числа, то `scanf` не закончится, а будет ожидать повторного ввода.

Более гибкий ввод обеспечивается чтением строки, например, функцией `gets`, а затем преобразованием ее в число при помощи функций типа `atoi`, `atol`, `atof` или `strtod`. Например:

```
#include <stdio.h>  
#include <stdlib.h>  
// . . .  
char s[81];  
gets(s);  
int i = atoi(s);
```

В таком варианте при нажатии клавиши Enter без предварительно введенного числа переменная `i` получит значение 0. Впрочем, можно перед использованием `atoi` провести анализ на ввод пустой строки.

Вывод символов на экран

Функция `puts` выводит строку символов на экран и переводит курсор на начало следующей строки. Если в программе не нужно переходить на новую строку, следует вос-

пользоваться функциями `fputs` или `printf`. Следующие инструкции выводят строку `s` без перехода на новую строку.

```
fputs(s, stdout);  
printf("%s", s);
```

Функция `printf` также может использоваться для вывода значений в указанном формате (см. описание функции `printf`).

В некоторых системах вывод на экран не происходит до появления в строке символа перевода строки. Чтобы все-таки заставить вывести данные на экран, можно использовать функцию `fflush`:

```
printf("Введите число:");  
fflush(stdout);
```

Сравнение консольного и файлового ввода/вывода

Консольные функции не согласуются с функциями файлового ввода/вывода. Так, функция `puts` автоматически добавляет символ новой строки, а функция `fputs` этого не делает. В следующем примере делается попытка вывести на экран содержимое текстового файла, но вывода не происходит, так как между строками исходного файла выводятся пустые строки:

```
#include <stdio.h>  
#define LINESZ 256  
int main()  
{  
    FILE *inf;  
    char s[LINESZ];  
    inf = fopen("test.txt", "r");  
    if (inf == NULL)  
    {  
        puts("Не удастся открыть входной файл");  
        return 1;  
    }  
    fgets(s, LINESZ, inf);  
    while (!feof(inf))
```



```
    {  
        puts(s);  
        fgets(s, LINESZ, inf);  
    }  
    return 0;  
}
```

Здесь ключевым местом являются инструкции:

```
while (!feof(inf))  
{  
    puts(s);  
    fgets(s, LINESZ, inf);  
}
```

Проблема заключается в том, что функция `fgets` помещает символ новой строки в конец прочитанной строки, а функция `puts` добавляет еще один символ новой строки. Одним из способов решения этой проблемы является использование функции `fputs` с указанием потока `stdout`. В этом случае вывод будет выполняться на экран, но лишних пустых строк уже не будет:

```
while (!feof(inf))  
{  
    fputs(s, stdout);  
    fgets(s, LINESZ, inf);  
}
```

В файле заголовков `stdio.h` определено несколько стандартных потоков ввода/вывода:

- `stdin` — стандартный ввод (по умолчанию — клавиатура);
- `stdout` — стандартный вывод (по умолчанию — экран);
- `stderr` — стандартный вывод сообщений об ошибках (по умолчанию — экран).

В этом списке делается различие между потоками `stdout` и `stderr`, и это правильно. Например, при перенаправлении вывода в файл (в UNIX или DOS) поток `stderr` по-прежнему будет выводить сообщения об ошибках на экран.

Функции проверки категории символов

Прототипы:

```
bool isalnum(int ch);  
bool isalpha(int ch);  
bool iscntrl(int ch);  
bool isdigit(int ch);  
bool isgraph(int ch);  
bool islower(int ch);  
bool isprint(int ch);  
bool ispunct(int ch);  
bool isspace(int ch);  
bool isupper(int ch);  
bool isxdigit(int ch);
```

Файл заголовков: `ctype.h` или `cctype`.

Позволяет проверить принадлежность символа к определенной категории. В файле заголовков `ctype.h` (или `cctype`) определены также функции: `toupper` и `tolower`.

 **ПРИМЕЧАНИЕ** В связи с отсутствием в C и ранних версиях C++ типа `bool` все функции, перечисленные выше, возвращали значение типа `int`.

Синтаксис

Каждая функция выделяет младший байт параметра `ch`, имеющего тип `int`. Значение функции имеет тип `bool`, означающий «ИСТИНА» (1) или «ЛОЖЬ» (0):

- `isalnum (ch)` проверяет, является ли символ буквой или цифрой;
- `isalpha (ch)` проверяет, является ли символ буквой;
- `iscntrl (ch)` проверяет, является ли символ управляющим символом (типа `line feed`, `del`, табуляции и т. п.). Большинство из этих символов находится в диапазоне `0x01–0x1F` (для кодировки ASCII);
- `isdigit (ch)` проверяет, является ли символ цифрой;

- `isgraph (ch)` проверяет, является ли символ видимым, то есть он не является символом пробела, табуляции и т. д.;
- `islower (ch)` проверяет, является ли символ буквой нижнего регистра;
- `isprint (ch)` проверяет, является ли символ печатаемым (`isgraph` + пробел);
- `ispunct (ch)` проверяет, является ли символ символом пунктуации, то есть печатаемым, но не буквой, не цифрой, не пробелом;
- `isspace (ch)` проверяет, является ли символ символом пробела или табуляции, или символом новой строки, или символом новой страницы (символом перевода формата);
- `isupper (ch)` проверяет, является ли символ буквой верхнего регистра;
- `isxdigit (ch)` проверяет, является ли символ символом шестнадцатеричной цифры (цифры, а также буквы от А до F в нижнем или верхнем регистрах).



ПРИМЕЧАНИЕ Несмотря на то что эти функции получают целый параметр, используется только его младший байт. Таким образом, подразумевается, что функции получают однобайтное значение — символ. Такой подход применен по той причине, что при некоторых параметрах компиляции в C/C++ байтовые величины в любых выражениях автоматически расширяются до целого значения.

В следующем примере печатается сообщение, если 5-й символ строки `s` является пробелом:

```
#include <ctype.h>
#include <iostream.h>
// . . .
char *s = "Here is a sample string";
if (isspace(s[4]))
    cout << "Character is space\n";
```


Функции работы с блоками памяти

Прототипы:

```
void *memchr(const void *p, int ch, size_t n);  
int memcmp(const void *p1, const void *p2, size_t n);  
void *memcpy(void *dest, const void *src, size_t n);  
void *memmove(void *dest, const void *src, size_t n);  
void *memset(const void *p, int ch, size_t n);
```

Файл заголовков: `string.h` или `cstring`.

Функции для эффективной работы с блоками памяти. В файле `string.h` также определены прототипы функций работы со строками. В старых версиях языка C функции работы с блоками определялись в файле заголовков `memory.h`.

В табл. 8 приведена краткая сводка функций работы с блоками памяти. В этой таблице указатели `p`, `p1` и `p2` являются указателями типа `void*`. Целый аргумент `n` обычно обозначает размер блока памяти — он может быть и меньше реального размера блока памяти, например, если нужно скопировать только часть блока памяти. Аргумент `ch` типа `int` представляет собой символ.

Таблица 8. Функции работы с блоками памяти

Функция	Действие
<code>memchr(p, ch, n)</code>	Возвращает указатель на первое вхождение байта, представленного младшим байтом аргумента <code>ch</code> в блоке памяти <code>p</code> длиной <code>n</code>
<code>memcmp(p1, p2, n)</code>	Сравнивает два блока памяти и возвращает значение: меньше нуля, равное нулю или больше нуля — аналогично кодам возврата функции <code>strcmp</code>
<code>memcpy(p1, p2, n)</code>	Копирует блок памяти длиной <code>n</code> байт из адреса <code>p2</code> по адресу <code>p1</code>

продолжение ➤

Таблица 8. Функции работы с блоками памяти (продолжение)

Функция	Действие
<code>memmove(p1, p2, n)</code>	Аналогична <code>memcpy</code> , но блоки <code>p1</code> и <code>p2</code> могут перекрываться
<code>memset(p, ch, n)</code>	Заполняет блок памяти символом, взятым из младшего байта <code>ch</code>

Нужно отметить, что все указатели этих функций объявлены как `const void *`, за исключением первых аргументов функций `memcpy` и `memmove`, которые объявлены просто как `void*`. Ключевое слово `const` говорит о том, что функции не изменяют данных, адрес которых передается через этот указатель (функция `memmove` кажется исключением, но реально, в случае перекрытия блоков памяти, данные модифицируются не через указатель `p2`, а через указатель `p1`).

В каждой функции аргумент `n` имеет тип `size_t`. Этот тип определен в файле `stdlib.h` и представляет собой достаточно большое по размеру целое (как правило — `unsigned long`).



ПРИМЕЧАНИЕ Большинство из этих функций возвращает указатель. Каждая из функций `memcpy`, `memmove` и `memset` возвращает свой первый аргумент — указатель на буфер (его удобно использовать в сложных выражениях). Функция `memchr` возвращает указатель на первый найденный байт. Так как возвращается указатель типа `void*`, то перед использованием он должен быть приведен к требуемому типу.

Пример:

```
#include <string.h>
//..
char *p, *buf = "This is a character X array.";
p = (char *)memchr(buf, 'X', strlen(buf));
int index = p - buf; // индекс в строке
```

Функции `memcpy` и `memmove` похожи друг на друга, но `memcpy` обычно быстрее, хотя ее поведение не определено при перекрывающихся блоках памяти. С другой стороны, функция

`memmove` более гибкая и может использоваться для сдвига значений внутри массива. Вот пример того, как можно перенести элементы массива с 1 по 100 в позиции от 0 до 99:

```
short arr[500];  
//..  
memmove(arr, arr+1, 100*sizeof(short));
```

Функции работы с датой и временем

Прототипы:

```
char *asctime(const struct tm *ctm);  
clock_t clock(void);  
char *ctime(time_t *t);  
double difftime(time_t t2, time_t t1);  
const struct tm gmtime(const time_t *t);  
struct tm localtime(time_t *t);  
struct tm mktime(time_t *t);  
struct tm strftime(char *s, size_t size, fmt, const  
    struct tm *ctm);  
time_t time(time_t *t);
```

Файл заголовков: `time.h` или `ctime`.

Стандартная библиотека C/C++ включает в себя ряд функций работы с датой и временем: получение и установка системного времени, представление времени в различных форматах, а также функции для получения времени исполнения программы.

Обзор

В приведенном ниже списке представлены функции работы с временем и датой. Аргументы `t`, `t1` и `t2` имеют тип `time_t` — числовое представление времени, а `struct tm` — специальная структура, в которой дата и время представлены в «расщепленном» виде (подробнее она будет описана ниже):

- `asctime(*tm)` возвращает строку, представляющую `tm`;
- `clock()` возвращает время исполнения программы как значение типа `clock_t` (это значение может быть пере-

ведено в секунды делением его на predeterminedенную в файле `time.h` константу `CLOCK_PER_SEC`);

- `ctime(*t)` возвращает строку, представляющую дату и время из параметра `t`;
- `difftime(t2,t1)` возвращает разницу времен (`t2-t1`) в секундах;
- `gmtime(*t)` возвращает указатель на временную структуру типа `tm`, инициализированную на основе даты/времени из `t`. Время устанавливается по Гринвичу;
- `localtime(*t)` аналогична функции `gmtime`, но время — локальное;
- `mktime(*t)` заполняет поля дня недели и дня года на основе имеющейся в `t` информации;
- `strftime(s,size,fmt,*tm)` возвращает отформатированную строку с датой и временем на основе формата `fmt`. Значение функции имеет тип `time_t`, соответствующий типу `tm`;
- `time(*t)` возвращает текущие дату/время в виде `time_t`. Если `*t` не равно `NULL`, то значение заносится и по этому адресу.

Все три специальных типа `tm`, `time_t` и `clock_t` определены в файле заголовков `time.h`. Тип `time_t` обычно представляется целым типом достаточного размера.



ПРИМЕЧАНИЕ Большинство действий с функциями работы с датой/временем начинается с получения текущей даты/времени путем вызова функции `time` (значение типа `time_t`).

Пример получения текущей даты/времени:

```
#include <time.h>
//..
time_t t;
t = time(NULL);
```

Простейший способ привести это значение к удобной форме — вызвать функцию `ctime`, которая возвращает строку вида:

день_недели месяц день часы:минуты:секунды год\n

Следующим образом можно вывести текущую дату и время:

```
#include <time.h>
#include <iostream.h>
//...
time_t t = time(NULL);
cout << ctime(&t);
```

Пример вывода предыдущего фрагмента:

Mon Mar 27 05:04:55 2000

Использование структуры tm

Альтернативный способ работы со временем — создание структуры tm при вызове одной из функций gmtime или localtime. В следующем примере в структуре типа tm получают текущие дату и время:

```
time_t t = time(NULL);
struct tm *tmp = localtime(&t);
```



ПРИМЕЧАНИЕ Структура, на которую указывает указатель tmp из предыдущего примера, будет перезаписываться при каждом вызове функций gmtime или localtime, поэтому, если значение необходимо сохранить для последующих операций, нужно сделать его копию: struct tm tmNow = *tmp;.

Структура типа tm замечательна тем, что в ней дата и время хранятся в «разбитом» виде, то есть число месяца, номер месяца и т. д. хранятся в разных данных (полях). Вот как эта структура определена в файле time.h:

```
struct tm
{
    int tm_sec; // секунды (0-59)
    int tm_min; // минуты (0-59)
    int tm_hour; // часы (0-23)
    int tm_mday; // день месяца (1-31)
    int tm_mon; // номер месяца (0-11)
    int tm_year; // номер года - 1900
```



```
int tm_wday; // день недели (0-6)
int tm_yday; // день года (0-365)
int tm_isdst; // признак летнего времени:
                // 0 – нет, > 0 – да
};
```

Далее можно использовать отдельные поля для своих целей. Например, выведем текущий день месяца:

```
time_t t = time(NULL);
struct tm *tmp = localtime(&t);
cout<<" Day of month is " << tmp->tm_mday;
```

Функция `mktime` получает указатель на структуру типа `tm` и подготавливает поля `tm_wday` и `tm_yday` в ней на основе значений других полей. Пример получения дня недели на основе номера месяца, дня месяца и года:

```
#include <time.h>
int get_weekday(int month, int day, int year)
{
    struct tm t;
    t.tm_mon = month - 1;
    t.tm_day = day;
    t.tm_year = year - 1900;
    if (mktime(&t) == -1)
        return -1; // ошибка преобразования
    return t.tm_wday;
}
```

Функция `mktime` возвращает значение `-1` в том случае, если не удастся правильно интерпретировать сформированные поля в структуре `tm`.

Функция `asctime` позволяет преобразовать поля структуры типа `tm` в легко читаемый вид: она формирует строку, аналогичную формируемой в функции `ctime`.

Сложное формирование даты/времени

Если есть необходимость сформировать более точное представление даты/времени, то нужно использовать функцию

strftime (см. описание функции strftime). Здесь рассмотрим пример получения дня недели:

```
#include <time.h>
#include <iostream.h>
//..
time_t t = time(NULL);
struct tm *tmp = localtime(&t);
char s[100];
strftime(s, 99, "The day of the week is %A.", tmp);
puts(s);
```

Функции работы со строками

Прототипы:

```
char *strcat(char *s1, char *s2)
char *strcmp(char *s1, char *s2)
char *strcpy(char *s1, char *s2)
size_t strlen(char *s)
char *strncat(char *s1, char *s2, size_t n)
int strncmp(char *s1, char *s2, size_t n)
char *strncpy(char *s1, char *s2, size_t n)
char *strstr(char *s1, char *s2)
```

Реже используемые функции:

```
char *strchr(char *s, int ch)
char *strcoll(char *s1, char *s2)
size_t strcspn(char *s1, char *s2)
char *strerror(size_t n)
char *strpbrk(char *s1, char *s2)
char *strrchr(char *s, int ch)
size_t strspn(char *s1, char *s2)
char *strtok(char *s1, char *s2)
size_t strxfrm(char *s1, char *s2, size_t n)
```

Файл заголовков: string.h или cstring.

Синтаксис

Стандартная библиотека C/C++ содержит большое количество функций работы со строками, в приведенном ниже

списке собраны наиболее популярные. В этом списке аргументы `s`, `s1` и `s2` являются указателями типа `char*`, а `n` — целое типа `size_t`. Если не оговорено особо, то каждая функция возвращает посредством указателя свой первый аргумент:

- `strcat(s1, s2)` добавляет `s2` к `s1`;
- `strcmp(s1, s2)` сравнивает строки и возвращает отрицательное (если `s1` меньше `s2`), нулевое (если `s1` равно `s2`) или положительное (если `s1` больше `s2`) значение;
- `strcpy(s1, s2)` копирует `s2` в `s1`;
- `strlen(s)` возвращает длину строки (без учета символа завершения строки);
- `strncat(s1, s2, n)` добавляет `s2` к `s1`, но не более `n` символов;
- `strncmp(s1, s2, n)` аналогична `strcmp`, но сравнивает не более `n` символов;
- `strncpy(s1, s2, n)` копирует `s2` в `s1`, но не более `n` символов;
- `strstr(s1, s2)` поиск первого вхождения подстроки `s2` в строку `s1`.

В следующем списке собраны оставшиеся, менее популярные, функции работы со строками. Здесь появляется еще один тип аргумента — целое `ch`, в котором используется только младший байт:

- `strchr(s, ch)` возвращает указатель на первое вхождение символа `ch` в строку `s`, если его нет, то возвращает `null`;
- `strcoll(s1, s2)` аналог `strcmp`, но учитывает установки локализации (см. функцию `setlocale`);
- `strcspn(s1, s2)` возвращает значение индекса любого из символов из `s2` в строке `s1`;
- `strerror(n)` возвращает указатель на строку с описанием ошибки номер `n`;

- `strpbrk(s1, s2)` аналогична `strcspn`, но возвращает указатель, а не индекс;
- `strrchr(s, ch)` аналогична `strchr`, но поиск ведется с конца строки;
- `strspn(s1, s2)` возвращает индекс первого символа в `s1`, отсутствующего в `s2`;
- `strtok(s1, s2)` возвращает следующую лексему из `s1`, отделенную любым символом из набора `s2`;
- `strxfrm(s1, s2, n)` преобразует строку из `s2` и помещает ее в `s1` на основе текущей локализации. Преобразуется не более `n` символов.



ПРИМЕЧАНИЕ Наиболее популярными функциями из этого набора являются `strcpy`, `strcat`, `strcmp` и `strlen`. Однако использование функций `strcpy` и `strcat` кроет в себе опасность переполнения приемного буфера, так как никаких проверок на соответствие размера буфера количеству записываемых символов не проводится.

В следующем примере буфер `message` будет переполнен:

```
#include <string.h>
//..
char message[12];
strcpy(message, "Hello, ");
strcat(message, " ");
strcat(message, "world!");
```

Чтобы избежать такой опасности, рекомендуется использовать аналогичные по свойствам функции `strncpy` и `strncat`, имеющие защиту от переполнения буфера:

```
#include <string.h>
#include <iostream.h>
//..
char message[12];
strncpy(message, "Hello, ", 12);
strncat(message, " ", strlen(message));
strncat(message, "world!", strlen(message));
cout << message << '\n';
```


В этом случае будет выведено:

Hello, world,

то есть лишние символы в строку не записываются.

Функция `strstr` используется для поиска и индексирования. Она возвращает указатель на подстроку, найденную в строке `s1`. Если нужен индекс, то можно использовать арифметику указателей:

```
#include <string.h>
//..
int index;
char str[] = "This land, this realm, this England";
index = strstr(str, "England")-str;
```

Все функции поиска и индексирования являются чувствительными к регистру символов.

Функция `strtok` используется для получения лексем — строк, отделенных символами-разделителями. Так, можно с ее помощью разбить предложение, записанное на «человеческом» языке, на отдельные слова, используя в качестве разделителей символы пробела и знаки пунктуации. При первом вызове `strtok` необходимо передать анализируемую строку в качестве аргумента `s1`; при последующем разборе вместо `s1` необходимо указывать `NULL`. При исчерпании всех лексем `strtok` возвратит `NULL`:

```
#include <string.h>
#include <iostream.h>
//..
char str[] = "This land, this realm, this England";
char *p = strtok(str, ",.");
while (p)
{
    cout << p << '\n';
    p = strtok(NULL, ",.");
}
```

При исполнении этого примера будет выведено:

This
land


```
this  
realm  
this  
England
```

Типы и возвращаемые значения

Некоторые функции имеют параметр типа `size_t`, определенный в файле `stdlib.h` как целое достаточно большого размера (обычно `unsigned long`). Строковые функции, возвращающие целые, имеют также тип значения `size_t`.

За исключением перечисленных случаев, каждая функция работы со строками возвращает указатель на свой первый аргумент. Такое решение позволяет строить достаточно сложные выражения:

```
strcat(strcat(strcpy(name, s1), s2), s3);
```

Функции файлового ввода/вывода

Прототипы функций открытия и закрытия файлов:

```
int fclose(FILE *f);  
FILE *fopen(const char *fname, const char *mode);  
FILE *freopen(const char *fname, const char *m, FILE *f);  
FILE *tmpfile(void);
```

Прототипы функций вывода в файл:

```
int fflush(FILE *f);  
int fprintf(FILE *f, const char *fmt, ...);  
int fputc(int ch, FILE *f);  
int fputs(const char *s, FILE *f);  
size_t fwrite(const void *p, size_t size, size_t n, FILE  
*f);  
int putc(int ch, FILE *f);
```

Прототипы функций чтения из файла:

```
int fgetc(FILE *f);  
char *fgets(char *s, int n, FILE *f);  
size_t fread(void *p, size_t size, size_t n, FILE *f);  
int fscanf(FILE *f, const char *fmt, ...);
```



```
int getc(FILE *f);  
int ungetc(int ch, FILE *f);
```

Прототипы функций для работы с позицией в файле:

```
int feof(FILE *f);  
int fgetpos(FILE *f, fpos_t *pos);  
int fseek(FILE *f, long off, int org);  
int fsetpos(FILE *f, const fpos_t *pos);  
long int ftell(FILE *f);
```

Прототипы функций обработки ошибок:

```
void clearerr(FILE *f);  
int ferror(FILE *f);  
void rewind(FILE *f);  
void setbuf(FILE *f, char *p);  
setvbuf(FILE *f, char *p, int mode, size_t size);
```

Файл заголовков: `stdio.h` или `cstdio`.

Обзор

Ниже перечислены некоторые функции, предназначенные для открытия и закрытия файлов. В этих функциях аргумент `f` типа `FILE*` является указателем на файловый поток ввода/вывода (в этом пункте для краткости будем называть его просто поток), а `fname` — строка, содержащая имя файла:

- `fclose(f)` закрывает поток ввода/вывода `f`;
- `fopen (fname, mode)` открывает файл с именем `fname` для работы в режиме, указанном строкой `mode`. При успешном открытии возвращается указатель на поток (таким образом полученный поток связывается с файлом), в противном случае — `NULL` (см. также описание функции `fopen`);
- `freopen (fname, m, f)` аналогична `fopen`, но предварительно закрывает поток `f`;
- `tmpfile()` открывает временный файл в режиме двоичного ввода/вывода и возвращает указатель на поток (см. также описание функции `tmpfile`).

После того как получен указатель на поток, он может использоваться как аргумент *f* в функциях ввода/вывода. Следующие функции предназначены для вывода:

- `fflush(f)` очищает буфер вывода посредством немедленной отправки данных для записи на физическое устройство. При успешном завершении возвращает значение 0, иначе возвращается значение EOF;
- `fprintf (f,fmt,...)` записывает в поток *f* переменные, список которых обозначен многоточием (...), в формате, указанном строкой *fmt*. Возвращает число записанных символов (см. описание функции `printf`);
- `fputc (ch,f)` записывает символ *ch* в поток *f*. При ошибке возвращает значение EOF, иначе — записанный символ;
- `fputs (s,f)` записывает строку символов *s* в поток *f*. Символ конца строки в файл не записывается. При ошибке возвращает значение EOF, иначе — неотрицательное число;
- `fwrite (p,size,n,f)` записывает *n* элементов длиной *size* байт из буфера, заданного указателем *p*, в поток *f*. Возвращает число записанных элементов;
- `putc (ch,f)` — то же, что и `fputc`.

Следующие функции считывают данные из файла:

- `fgetc (f)` возвращает очередной символ из потока *f*. Если символ не может быть прочитан, то возвращается значение EOF. Символ возвращается как `int`;
- `fgets (s,n,f)` читает не более *n*-1 байт из потока *f* в строку *s*, прекращая чтение при обнаружении символа новой строки или конца файла. Символ новой строки при чтении не отбрасывается, а помещается в конец строки. Прочитанная строка дополняется ограничителем строки (нуль-терминатором `'\0'`). При обнаружении ошибки возвращается `NULL`, в противном случае — указатель на строку *s* (конец файла в данном контексте ошибкой не является);

- `fread (p, size, n, f)` читает `n` элементов, каждый длиной `size` байт, в буфер, заданный указателем `p`, из потока `f`. Возвращает число прочитанных элементов;
- `fscanf (f, fmt, ...)` в соответствии с форматом, указанным строкой `fmt`, читает из потока `f` переменные, список адресов которых обозначен многоточием (...). Возвращает число прочитанных элементов или EOF в случае неудачного чтения (см. описание функции `scanf`);
- `getc (f)` — то же, что и `fgetc`;
- `ungetc (ch, f)` возвращает символ `ch` обратно во входной поток `f`. Возвращает EOF в случае ошибки, в противном случае возвращает `ch`. Гарантируется возвращение только одного символа.

Некоторые функции определяют позицию в файле. Они особенно удобны для организации произвольного доступа к файлам. Аргумент `pos` имеет тип `fpos_t`, который определяется в файле `stdio.h`, `fpos_t` — это либо целое, либо структура достаточно большая для того, чтобы записать или сохранить любую позицию в файле:

- `feof (f)` возвращает ненулевое (истинное) значение при обнаружении конца файла (макрос);
- `fgetpos (f, *pos)` возвращает текущую позицию в файле, связанном с потоком `f`, и копирует значение по адресу `pos`. Это значение позднее может использоваться функцией `fsetpos`. Возвращаемое значение имеет тип `fpos_t`;
- `fseek (f, off, org)` перемещает текущую позицию в файле, связанном с потоком `f`, на позицию `off`, отсчитываемую от значения `org` (целое);
- `fsetpos (f, pos)` перемещает текущую позицию в файле, связанном с потоком `f`, на позицию `*pos`, предварительно полученную с помощью функции `fgetpos`;
- `ftell (f)` возвращает текущую позицию в файле, связанном с потоком `f`, как длинное целое.

Для функции `fseek` аргумент `org` может принимать одно из следующих значений: `SEEK_SET` (установить позицию

относительно начала файла), `SEEK_CUR` (установить позицию относительно текущей позиции) и `SEEK_END` (установить позицию относительно конца файла).

Остальные функции связаны с обработкой ошибок и пользовательскими буферами. Аргумент `p` имеет тип `char*`. C/C++ по умолчанию обеспечивают файловые потоки своими буферами размером `BUFSIZ` для минимизации числа обращений к диску и сокращения времени чтения/записи:

- `clearerr (f)` очищает флаги ошибок для `f`. После возникновения ошибки работа с файлами может быть продолжена только после сброса ошибки с помощью функций `clearerr` или `rewind`;
- `ferror (f)` возвращает целое, означающее код ошибки. 0 означает отсутствие ошибки;
- `rewind (f)` очищает флаги ошибок и устанавливает текущую позицию на начало файла;
- `setbuf (f, p)` устанавливает пользовательский буфер ввода/вывода, заданный указателем `p`. Размер буфера должен быть длиной `BUFSIZ`. Если `p` является нулевым указателем, то буферизация отменяется;
- `setvbuf (f, p, mode, size)` аналогична `setbuf`, за исключением того, что `mode` указывает режим, а `size` — размер буфера. Возвращаются значение 0 при успешном завершении и ненулевое значение в случае ошибки. Параметр `mode` является целым, принимающим одно из значений: `_IOFBF` (полная буферизация), `_IONBF` (запрет буферизации) и `_IOLBF` (для вывода — построчная буферизация, то есть опустошение буфера при записи в буфер символа новой строки).

Чтение и запись текстовых файлов

Перед чтением или записью файл необходимо открыть. Это можно сделать с помощью функции `fopen`. При успешном завершении она возвращает указатель типа `FILE*`, который в дальнейшем используется в качестве параметра в других функциях:


```
#include <stdio.h>
// . . .
FILE *fp;
fp = fopen("test.txt", "w");
```

Здесь строка "w" является параметром, определяющим режим, в котором будет использоваться поток и связанный с ним файл. В данном случае строка "w" указывает, что поток используется для вывода в текстовом режиме (см. описание функции `fopen`).

Различие между текстовым и двоичным режимами достаточно мало. В текстовом режиме каждая пара символов carriage return/line feed (возврат каретки/перевод строки) при чтении из файла в памяти заменяется на один символ line feed (символ новой строки), и наоборот, при выводе символ новой строки превращается в пару символов (это относится только к операционным системам семейства DOS и Windows; в UNIX символ carriage return вообще не используется, а в Mac символ line feed заменяется на символ carriage return, и наоборот).

Основным различием между текстовым и двоичным режимами является способ интерпретации информации. В текстовом файле числа представляются символами цифр, а в двоичном используется внутреннее представление числа, которое зависит от типа компьютера:

```
#include <stdio.h>
// . . .
void test_files(void)
{
    FILE *fp;
    int i = 12;
    fp = fopen("test.txt", "w");
    if (fp != NULL)
        fprintf(fp, "Вот число: %d\n", i);
    else
        fputs("Ошибка открытия файла\n", stderr);
}
```


При работе с текстовыми файлами отличным инструментом являются функции `fgets` и `fputs` — они согласованно работают друг с другом. Функция `fgets` читает строку из файла и включает в нее символ новой строки и нуль-терминатор. Функция `fputs` выводит такую строку без добавления символа новой строки и нуль-терминатора.

Рассмотрим пример программы, копирующей текстовый файл. Каждая строка в файле-копии начинается символом «>»:

```
#include <stdio.h>
#define LINESZ 256
void main(void)
{
    FILE *inf, *outf;
    char s[LINESZ];
    inf = fopen("test.txt", "r");
    outf = fopen("test1.txt", "w");
    if (inf == NULL || outf == NULL)
    {
        puts("Не удалось открыть файл.\n");
        return 1;
    }
    fgets(s, LINESZ, inf);
    while (!feof(inf))
    {
        fputs(">", outf);
        fputs(s, outf);
        fgets(s, LINESZ, inf);
    }
}
```

Чтение и запись двоичных файлов

Чтение и запись двоичных файлов обычно производятся с помощью функций `fread` и `fwrite`.

Чтобы работать с двоичным файлом, необходимо использовать символ `b` в параметре, определяющем режим открытия (см. также описание функции `fopen`). В двоичном ре-

жине никаких преобразований данных не производится, то есть нуль-терминатор при чтении не добавляется:

```
#include <stdio.h>
// . . .
FILE *fb;
fb = fopen("stuff.dat", "wb");
```

Ниже приводится текст программы вывода содержимого файла в шестнадцатеричном формате (дампа файла). Эта программа запрашивает и получает имя файла, а затем выводит дамп на стандартное устройство вывода:

```
#include <stdio.h>
#include <ctype.h>
#define BYTES_PER_ROW 16
int main(void)
{
    FILE *fb;
    char filespec[256], buf[BYTES_PER_ROW];
    int i, n;
    printf("Введите имя файла:");
    gets(filespec);
    fb = fopen(filespec, "rb");
    if (fb == NULL)
    {
        printf("Не удалось открыть файл %s\n", filespec);
        return 1;
    }
    do
    {
        n = fread(buf, 1, BYTES_PER_ROW, fb);
        for (i = 0; i < n; i++)
            // шестнадцатеричный дамп
            printf("%02X ", buf[i]);
        for (i = 0; i < n; i++)
            // символьный дамп
            if (iscntrl(buf[i]))
                putchar('.');
            else
```



```
        putchar(buf[i]);
    putchar('\n');
}
while (n == BYTES_PER_ROW);
return 0;
}
```

Ключевая инструкция этой программы:

```
n = fread(buf, 1, BYTES_PER_ROW, fb);
```

Параметр `buf` определяет массив, в который заносятся считываемые данные. Второй параметр задает размер каждого считываемого элемента. Так как этот размер равен 1, то функция считывает байты по одному. Третий параметр задает число элементов (в нашем случае байт) чтения. Здесь он определен константой `BYTES_PER_ROW`, равной 16, то есть длине одной строки дампа. После чтения программа выводит шестнадцатеричные значения байт и их символьные представления.



СОВЕТ При определении размера указанного типа данных может быть полезен оператор `sizeof`.

Так как функция `fread` возвращает число реально считанных элементов, то при числе считанных символов меньше 16 байт (что возможно только при достижении конца файла) программа выводит эти символы и завершается:

```
do
{
    n = fread(buf, 1, BYTES_PER_ROW, fb);
    // . . .
}
while (n == BYTES_PER_ROW);
```

Функция `fwrite` работает аналогичным образом и имеет тот же список параметров — она пытается записать в файл заданное число элементов. Если размер элемента равен 1, то записываются отдельные байты:

```
n = fwrite(buf, 1, BYTES_PER_ROW, fb);
```


(Информация о функциях `gets`, `puts`, `printf` и `putchar` приведена в разделе «Функции консольного ввода/вывода».)

Сводный список функций

Далее в алфавитном порядке перечислены функции стандартной библиотеки C++.

abort

Прототип:

```
void abort(void);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Вызывает немедленную аварийную остановку программы. Программа завершается без выполнения необходимых в этом случае процедур, таких как очистка буферов ввода/вывода, закрытие файлов и т. п., что может привести к потере данных. Для нормального завершения используется функция `exit`. Последний вариант является более предпочтительным даже при аварийном завершении.

abs

Прототип:

```
int abs(int num);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Возвращает абсолютное значение целого числа `num`.

Абсолютное значение неотрицательного числа `num` — само число `num`, отрицательного `num` — противоположное ему положительное значение, то есть `-num`.

Пример использования:

```
#include <stdlib.h>
#include <iostream.h>
// . . .
cout << abs(-5); // выводится число 5
```

Для получения абсолютного значения длинного целого используется функция `labs`, для получения абсолютного значения вещественного числа — функция `fabs`.

acos

Прототип:

```
double acos(double x);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает значение арккосинуса x ($\arccos x$) в радианах и является функцией, обратной функции косинус. Параметр x может изменяться от -1 до 1 .

В следующем примере вычисляется значение арккосинуса 0:

```
#include <math.h>
#include <iostream.h>
// . . .
cout << "acos(0)=" << acos(0) << '\n';
```

См. также функции `asin`, `atan`, `atan2`, `cos`, `sin` и `tan`.

asin

Прототип:

```
double asin(double x);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает значение арксинуса параметра x ($\arcsin x$) в радианах и является функцией, обратной функции синус. Параметр x может изменяться от -1 до 1 .

В следующем примере вычисляется значение арксинуса 1:

```
#include <math.h>
#include <iostream.h>
// . . .
cout << "asin(1)=" << asin(1) << '\n';
```

См. также функции `acos`, `atan`, `atan2`, `cos`, `sin` и `tan`.

assert

Прототип:

```
void assert(int expression);
```

Файл заголовков: `assert.h` или `cassert`.

Вызывает немедленное завершение программы при условии, что значение выражения `expression` является ложным (нулевым).

Проверяет условие продолжения программы `expression`. Если его значение истинно (не равно 0), выполнение продолжается, в противном случае выводится сообщение об ошибке вида: `Assertion failed: expression, file <имя_файла>, line <номер_строки>`, где `expression` является параметром функции, и программа завершается.

В следующем примере программа завершается, если значение `n` превышает 100:

```
#include <assert.h>
// . . .
assert(n <= 100);
```

Если значение `n` больше 100, то программа завершается, напечатав следующую строку: `Assertion failed: n <= 100, file: Examine.cpp, line 25`.

Определение макроса `NDEBUG` запрещает выполнение всех макросов `assert` (см. также раздел «`#define`» в главе «Директивы, макросы и операторы препроцессора»). Это полезно для исключения из программы лишних строк после завершения отладки программы. Так как строки `assert` из текста не убираются, то всегда легко вернуться к отладочному режиму, удалив определение макроса `NDEBUG`.

atan

Прототип:

```
double atan(double x);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает значение арктангенса `x` (`arctg x`) в радианах и является функцией, обратной функции тангенс.

В следующем примере печатается значение арктангенса числа 5.2:

```
#include <math.h>
#include <iostream.h>
```



```
// . . .  
cout << "atan(5.2)=" << atan(5.2) << '\n';
```

См. также функции `acos`, `asin`, `atan2`, `cos`, `sin` и `tan`.



ПРИМЕЧАНИЕ Одним из интересных вариантов использования функции `atan` является получение числа «пи»: `double pi = atan(1) * 4;`

atan2

Прототип:

```
double atan2(double y, double x);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает значение арктангенса отношения параметров y/x ($\arctg(y/x)$) в радианах и является функцией, обратной функции тангенс. Как `atan`, так и `atan2` получают отношение катетов прямоугольного треугольника. Разница заключается в том, что `atan2` получает оба значения обеих сторон в явном виде, а не в виде отношения. В следующем примере печатается значение арктангенса угла треугольника при катетах 108.3 и 42:

```
#include <math.h>  
#include <iostream.h>  
// . . .  
cout << atan2(108.3, 42.0) << '\n';
```

См. также функции `acos`, `asin`, `atan`, `cos`, `sin` и `tan`.

atexit

Прототип:

```
void atexit(void (*func)(void));
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Регистрирует функцию `func`, которая будет вызываться при завершении программы. Функция завершения `func` может быть полезна для того, чтобы собрать все завершающие действия в одном месте. При этом команды завершения программы могут быть в любой точке. Функция `func` не должна иметь ни параметров, ни возвращаемого значения.

Следующий код регистрирует функцию, которая выводит прощальное сообщение:

```
#include <stdlib.h>
void term_func(void)
{cout << "Поздравляем с успешным завершением!\n"
}
// . . .
atexit(&term_func);
```

Если программа завершается вызовом функции `abort`, то функция `func` не выполняется. При завершении программы любым другим способом (даже если это `exit(EXIT_FAILURE)`) вызывается функция `func`. По стандарту ANSI можно зарегистрировать до 32 функций завершения (в некоторых реализациях может быть больше), причем при завершении программы функции завершения будут выполняться по принципу: «первый пришел — последний ушел».

atof

Прототип:

```
double atof(const char *str);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Преобразует строку, отображающую в виде символов целое или вещественное число, в соответствующее вещественное число.

При разборе строки функция пропускает возможные пробелы, стоящие в начале строки, и обрабатывает последующие символы до тех пор, пока они могут формировать вещественное число. Остальные символы игнорируются. Например, `atof("-24.56xy7")` равно `-24.56`. Функция воспринимает в качестве показателя степени 10 символы, стоящие после символов «E» или «e», то есть запись «5E2» воспринимается как «5*10²» или 500. Если ни один символ в строке не воспринят, то в большинстве реализаций функция `atof` возвращает значение 0.0.

В следующем примере с клавиатуры считывается число и сохраняется в переменной `x`. Предполагается, что максимальная длина строки составляет 80 символов:


```
#include <stdlib.h>
#include <stdio.h>
// . . .
char input_str[81];
double x = atof(gets(input_str));
```

См. также функции `atoi`, `atol`.

atoi

Прототип:

```
int atoi(const char *str);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Преобразует строку, отображающую в виде символов целое число, в соответствующее целое число.

При разборе строки функция пропускает возможные пробелы в начале строки и обрабатывает последующие символы до тех пор, пока они могут формировать целое число. Остальные символы игнорируются. Например, `atoi("23.9x")` равно 23. Если ни один символ строки не воспринят, то в большинстве реализаций функция `atoi` возвращает значение 0.

В следующем примере с клавиатуры считывается число и сохраняется в переменной `n`. Предполагается, что максимальная длина строки составляет 80 символов:

```
#include <stdlib.h>
#include <stdio.h>
// . . .
char input_str[81];
int n = atoi(gets(input_str));
```

См. также функции `atol`, `atof`, `strtol` и `strtoul`.

atol

Прототип:

```
long atol(const char *str);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Преобразует строку, отображающую в виде символов целое число, в соответствующее длинное целое значение.

При разборе строки функция пропускает возможные пробелы в начале строки и обрабатывает следующие символы до тех пор, пока они могут формировать целое число типа `long`. Остальные символы игнорируются. Например, `atol("23.9x")` равно `23L`. Если ни один символ в строке не воспринят, то в большинстве реализаций функция `atol` возвращает значение `0L`.

В следующем примере с клавиатуры считывается число и сохраняется в переменной `n`. Предполагается, что максимальная длина строки составляет 80 символов:

```
#include <stdlib.h>
#include <stdio.h>
// . . .
char input_str[81];
long int n = atol(gets(input_str));
```

См. также функции `atoi`, `atof`, `strtol` и `strtoul`.

bsearch

Прототип:

```
void bsearch(const void *key, const void *buf, size_t num,
size_t size, int (*compare)(const void *, const void *) );
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Производит поиск в массиве `buf` элемента, совпадающего с тем, на который указывает `key`, и возвращает указатель на найденный элемент. `num` — количество элементов в массиве `buf`, `size` — размер элемента массива в байтах. Массив должен быть отсортированным, например, с помощью функции `qsort`.

Функция обратного вызова `compare` (callback-функция) должна выполнять сравнение двух элементов массива и возвращать одно из трех значений `-1`, `0` или `1` в зависимости от того, меньше, равен или больше первый элемент в сравнении со вторым. Несмотря на то что оба параметра

функции `strcmp` определены как `void*`, реально параметрами функции являются указатели на элементы, и она должна приводить их к соответствующему типу перед разыменованием (получением значений по адресу).

Если функция нашла искомый элемент, то она возвращает указатель на него, в противном случае возвращается `NULL` (см. описание функции `qsort`).

В следующем примере сортируется массив целых чисел. Функция `cmp_int` получает указатели типа `void*`, которые она приводит к правильному типу, и затем определяет указываемые ими значения:

```
#include <stdlib.h>
#include <iostream.h>
int cmp_int(const void *p1, const void *p2)
{
    int i1 = *(int *)p1;
    int i2 = *(int *)p2;
    return i1 < i2 ? -1 : (i1 == i2 ? 0 : 1);
}
void main()
{
    int dat[] = { 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 };
    int i = 49;
    void *p =
    bsearch(&i, dat, sizeof(dat)/sizeof(int), sizeof(int),
        &cmp);
    if (p == NULL)
        cout << "Число в массиве не найдено\n";
    else
        cout << "Число в массиве найдено\n";
}
```

calloc

Прототип:

```
void *calloc(size_t num, size_t size);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Выделяет блок памяти (num элементов по size байт) и возвращает указатель на него. Каждый элемент полученного массива инициализируется нулевым значением (в отличие от функции malloc). Если выделить блок памяти не удалось, возвращается NULL.

В следующем примере выделяется блок памяти для сохранения n целых, причем каждый элемент получает значение 0:

```
#include <stdlib.h>
#include <iostream.h>
int n;
int *p;
// . . .
p = (int *)calloc(n, sizeof(int));
if (p == NULL)
{
    cout << "Не удалось выделить блок памяти.\n";
    exit(EXIT_FAILURE);
}
```

См. также функции free, malloc и realloc.

ceil

Прототип:

```
double ceil(double x);
```

Файл заголовков: math.h или cmath.

Округляет вещественное значение x до ближайшего большего целого и возвращает его как вещественное.

Рассмотрим пример функции, возвращающей случайные числа от 1 до n:

```
#include <math.h>
#include <stdlib.h>
int random_N(int n)
{
    return (int) ceil(n * rand()/RAND_MAX);
}
```

См. также функции floor и modf.

cos

Прототип:

```
double cos(double x);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает значение косинуса угла, равного x радиан. Изменяется от -1 до 1 .

В следующем примере печатается значение косинуса $\pi/2$. Результат может не совпадать с тем, который приводится в учебниках по математике, но значение должно быть близко к 0:

```
#include <math.h>
#include <iostream.h>
// . . .
double pi = atan(1.0) * 4;
cout << "cos(pi/2) = " << cos(pi/2) << '\n';
```

См. также функции `acos`, `asin`, `atan`, `atan2`, `sin` и `tan`.

cosh

Прототип:

```
double cosh(double x);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает значение гиперболического косинуса угла, равного x радиан. Если значение функции окажется вне представимого диапазона, то функция возвращает значение `HUGEVAL`, а глобальная переменная `errno` получает значение `ERANGE`.

См. также функции `sinh` и `tanh`.

div

Прототип:

```
div_t div(int numerator, int denominator);
```

Файл заголовков: `stdlib.h` или `stdlib`.

Функция `div` производит деление целого `numerator` на целое `denominator`, возвращая результат в структуре, содержащей частное и остаток:

```
struct div_t
{
    int quot; //частное
    int rem;  //остаток
};
```

См. также функцию `ldiv`, которая производит те же действия, но над длинными целыми.

Напомним, что остаток можно получить при помощи оператора деления по модулю (определения остатка от деления) — `%`.

exit

Прототип:

```
void exit(int exit_code);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Функция `exit` выполняет нормальное завершение программы. В результате все данные, еще не выведенные из буферов, выводятся в файлы, и запускаются все зарегистрированные функции завершения (см. также функцию `atexit`).

В файле `stdlib.h` определяются, по крайней мере, два значения для кода завершения: `EXIT_SUCCESS` (0 — при благополучном завершении) и `EXIT_FAILURE` (ненулевое значение — при неблагоприятном завершении). В некоторых реализациях могут быть добавлены и другие коды завершения.

Пример успешного завершения программы:

```
#include <stdlib.h>
exit(EXIT_SUCCESS);
```

exp

Прототип:

```
double exp(double x);
```


Файл заголовков: `math.h` или `cmath`.

Возвращает значение экспоненты параметра x , то есть возводит число e в указанную степень (см. также функцию `pow`).

Используя эту функцию, легко получить значение числа e :

```
#include <math.h>
...
double e = exp(1.0);
```

fabs

Прототип:

```
double fabs(double x);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает абсолютное значение x .

Абсолютное значение для неотрицательных чисел — это само значение, а для отрицательных — противоположное значение, то есть $-x$. Пример использования:

```
#include <math.h>
#include <iostream.h>
// . . .
cout << fabs(-5.07); // выводится 5.07
```

Для вычисления абсолютного значения целого числа используется функция `abs`, для абсолютного значения длинного целого — функция `labs`.

floor

Прототип:

```
double floor(double x);
```

Файл заголовков: `math.h` или `cmath`.

Округляет x до ближайшего меньшего целого.

Рассмотрим пример функции, возвращающей случайное целое в диапазоне от 0 до $n-1$:


```
#include <math.h>
#include <stdlib.h>
int random_0_to_N_1(int n)
{
    return (int)floor(n * rand()/RAND_MAX);
}
```

См. также функции `ceil` и `modf`.

fmod

Прототип:

```
double fmod(double x, double y);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает остаток от деления x на y . Другими словами, функция вычисляет наибольшее целое i , не превышающее значения x/y , а затем возвращает величину $x - i * y$. Это действие аналогично действию оператора остатка (`%`), но выполняется над вещественными числами и возвращает также вещественное число. Рассмотрим пример:

```
#include <math.h>
#include <iostream.h>
// . . .
cout << fmod(31,2) << '\n'; // выводит "1"
cout << fmod(31.05,2) << '\n'; // выводит "1.05"
cout << fmod(7.6,2.5) << '\n'; // выводит "0.1"
```

fopen

Прототип:

```
FILE *fopen(const char *filename, const char *mode);
```

Файл заголовков: `stdio.h` или `cstdio`.

Функция `fopen` (`file open`) открывает файл для чтения или записи либо того и другого вместе, и возвращает указатель на поток. Далее этот указатель используется в функциях чтения/записи. В параметре `filename` возможно указание имени папки, в которой находится файл.

Спецификация режима

Параметр `mode` является строкой, содержащей одно из значений, перечисленных в табл. 9.

Таблица 9. Режимы открытия файлов функцией `fopen`

mode	Действие
"r"	Открытие текстового файла на чтение
"w"	Создание нового текстового файла на запись
"a"	Открытие текстового файла для добавления
"rb"	Открытие двоичного файла на чтение
"wb"	Создание нового двоичного файла на запись
"ab"	Открытие двоичного файла для добавления
"r+"	Открытие текстового файла на чтение и запись
"w+"	Создание нового текстового файла на чтение и запись
"a+"	Открытие текстового файла на чтение и запись
"rb+"	Открытие двоичного файла на чтение и запись
"wb+"	Создание нового двоичного файла на чтение и запись
"ab+"	Открытие двоичного файла на чтение и запись

Как правило, текстовым называется файл, в котором данные представлены в коде ASCII.

Технически единственной разницей между текстовым и двоичным файлом является способ обработки символа конца строки. В текстовом режиме (для операционных систем DOS и Windows) при выводе каждый символ новой строки (`'\n'`) вызывает запись в файл двух символов `0x0D` и `0x0A`, а при вводе пара символов `0x0D` и `0x0A` превращается в один `'\n'`. В других операционных системах (типа UNIX) такого преобразования не происходит, так как в них используется только один символ для указания конца строки — таким образом, для них нет различия между текстовым и двоичным файлом.

В следующем примере читается имя файла с консоли и используется для открытия файла на чтение в текстовом режиме. Если функция `fopen` возвращает значение `NULL`, то файл не открывается:

```
#include <stdio.h>
#include <stdlib.h>
// . . .
FILE *fp;
char fspec[81];
printf("Введите имя файла: ");
gets(fspec);
fp = fopen(fspec, "r");
if (fp == NULL)
{
    printf("Не удалось открыть файл %s\n", fspec);
    exit(1);
}
// Далее операции с использованием fp...
```

Правила использования

Рассмотрим еще ряд правил использования функции `fopen` в различных режимах.

- Если функция не может открыть файл, она возвращает значение `NULL`.
- Если открывается файл на чтение или добавление, то этот файл должен существовать, иначе возвращается ошибка.
- Основное требование при открытии файла на запись — правильное указание имени файла в параметре `fname`. Если существует файл с указанным именем, то после открытия его содержимое стирается, а длина файла устанавливается в 0.
- При отсутствии файла во время его открытия на чтение/запись создается новый файл.
- Если файл был открыт в режиме чтения/записи, то нельзя переключаться между чтением/записью и на-

оборот без предварительной очистки буферов. Очистка буферов происходит при вызове функций `fflush`, `fseek`, `fsetpos` и `rewind` (см. также раздел «Функции файлового ввода/вывода»).

free

Прототип:

```
void free(void *pointer);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Освобождает блок памяти, полученный ранее одной из функций `malloc`, `calloc` или `realloc`. Попытка освободить блок памяти, полученный без помощи указанных функций, приведет к ошибке (вплоть до «зависания» компьютера). Операторы `new` и `delete` работают аналогично, кроме того, они выполняют и другие служебные действия. Функции `malloc` и `free` оставлены в библиотеке для совместимости с C.

В следующем примере сначала выделяется, а затем освобождается блок памяти для `n` символов:

```
#include <stdlib.h>
int n;
char *p;
// . . .
p = reinterpret_cast<char*>(malloc(n));
// . . .
free(p);
```

Несмотря на то что при использовании функции `malloc` необходимо явное приведение типа, для функции `free` явного приведения типа не нужно, так как параметр функции `free` имеет тип `void*`.

См. также функции `malloc`, `calloc` и `realloc`.

frexp

Прототип:

```
double frexp(double x, int *exp_ptr);
```


Файл заголовков: `math.h` или `cmath`.

Выделяет мантиссу и показатель степени числа x . Возвращает значение мантиссы и копирует экспоненциальную часть по адресу `exp_ptr`. Так как практически все современные компьютеры работают в двоичной системе счисления, то мантисса будет числом меньшим 1.0, а показатель экспоненты представляет собой степень 2.

Рассмотрим пример разложения числа 48.0 на мантиссу 0.75 и показатель степени 6 (2 в степени 6 равно 64, а $0.75 \cdot 64 = 48$):

```
#include <math.h>
#include <iostream.h>
// . . .
int exp;
double mant = frexp(48.0, &exp);
cout << "mant = " << mant << endl;
// выводит строку: mant = 0.75
cout << "exp = " << exp << endl;
// выводит строку: exp = 6
```

getc

Прототип:

```
int getc(FILE *stream);
```

Файл заголовков: `stdio.h` или `cstdio`.

Считывает следующий символ из указанного потока ввода. Если больше нет доступных символов, то функция возвращает значение EOF.

Функция `getc` идентична функции `fgetc`. Причиной ее включения в библиотеку является то, что во многих реализациях функция `getc` определяется как макрос, а не как функция. Следовательно, при использовании `getc` в таких реализациях программа будет выполняться быстрее, чем при использовании функции `fgetc` (см. раздел «Функции файлового ввода/вывода»).

getchar

Прототип:

```
int getchar(void);
```

Файл заголовков: `stdio.h` или `cstdio`.

Считывает очередной символ со стандартного потока ввода, обычно связанного с клавиатурой. Как правило, для ввода с клавиатуры операционная система использует буфер, поэтому программа не получит символа до тех пор, пока не нажата клавиша Enter (см. раздел «Функции консольного ввода/вывода»).

getenv

Прототип:

```
int getenv(const char *var_name);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Возвращает указатель на строку, которая содержит описание указанной переменной окружения (среды).

Рассмотрим пример вывода на консоль переменной окружения `PATH`:

```
#include <stdlib.h>
#include <iostream.h>
// . . .
cout << "PATH = " << getenv("PATH") << '\n';
```

gets

Прототип:

```
char *gets(char *str);
```

Файл заголовков: `stdio.h` или `cstdio`.

Считывает строку до появления символа новой строки и помещает считанную строку по адресу, заданному указателем `str`. Возвращает указатель на считанную строку, то есть `str`. Необходимо помнить, что во избежание переполнения массив для приема строки должен быть достаточно размера. В противном случае безопаснее использовать

функцию `fgets`, так как в ней производится контроль за заполнением буфера. В отличие от функции `fgets` функция `gets` не копирует символ новой строки в буфер строки (см. раздел «Функции консольного ввода/вывода»).

labs

Прототип:

```
long int labs(long int num);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Возвращает абсолютное значение `num`. Абсолютное значение для неотрицательных чисел — это само значение, для отрицательных — противоположное значение, то есть `-num`.

Пример использования:

```
#include <stdlib.h>
#include <iostream.h>
// . . .
cout << labs(-5L) << endl; // выводится 5
```

См. также функции `abs` и `fabs`.

ldexp

Прототип:

```
double ldexp(double x, int exp);
```

Файл заголовков: `math.h` или `cmath`.

Получает мантиссу и показатель степени и возвращает число, равное произведению мантиссы на 2 в степени показатель степени. Противоположна функции `frexp` и выполняет практически те же действия, что и `pow`.

ldiv

Прототип:

```
ldiv_t ldiv(long numerator, long denominator);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Выполняет деление `numerator` на `denominator`, возвращая результат в структуре типа `ldiv_t`, содержащей частное и остаток.

Структура `ldiv_t` имеет следующий формат:

```
struct ldiv_t
{
    long quot; //частное
    long rem;  //остаток
};
```

Поведение функции не определено при нулевом значении `denominator`.

См. также функцию `div`, которая производит аналогичные действия над целыми.

 **ПРИМЕЧАНИЕ** Остаток можно получить при помощи оператора получения остатка от деления (%).

localeconv

Прототип:

```
struct lconv *localeconv(void);
```

Файл заголовков: `locale.h` или `clocale`.

Возвращает структуру, которая содержит детальное описание текущих установок локальной среды, то есть информацию, зависящую от страны. В частности, в этой структуре указывается символ-разделитель целой и дробной части числа (например, в США таковым является точка (.), а в России — запятая (,)). Пример использования:

```
#include <locale.h>
// . . .
struct lconv *lc = localeconv();
```

Значения полей этой структуры можно изменять только с помощью функции `setlocale`. Более подробную информацию об этой структуре можно найти в документации компилятора.

log

Прототип:

```
double log(double x);
```


Файл заголовков: `math.h` или `cmath`.

Возвращает значение натурального логарифма x ($\ln x$, логарифм по основанию числа « e ») и является обратной к функции `exp`. Если значение x меньше нуля, то возвращается значение NaN (Not a Number, не определено), если значение x равно нулю, то возвращается значение INF (Infinity, бесконечность).

См. также функции `log10` и `exp`.

log10

Прототип:

```
double log10(double x);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает значение десятичного логарифма x ($\lg x$, логарифм по основанию 10). Если значение x меньше нуля, то возвращается значение NaN (Not a Number, не определено), если значение x равно нулю, то возвращается значение INF (Infinity, бесконечность).

См. также функции `log` и `exp`.

longjmp

Прототип:

```
void longjmp(jmp_buf envbuf, int status);
```

Файл заголовков: `setjmp.h` или `csetjmp`.

Выполняет переход по адресу `envbuf`, ранее определенному с помощью функции `setjmp`. Эти функции позволяют выполнять переходы между функциями. Параметр `status` указывает новое значение адреса возврата для функции `setjmp`.

Действие функции `longjmp` заключается в переходе на инструкцию, содержащую функцию `setjmp`. Следовательно, функция `setjmp` исполняется заново, но выходным значением ее является значение `status`. Так как первичное исполнение функции `setjmp` завершается с кодом возврата

равным 0, то, установив значение `status`, отличное от 0, можно распознать, что управление передано в результате исполнения функции `longjmp`. Пример вызова функции `longjmp`:

```
#include <setjmp.h>
// . . .
jmp_buf envbuf;
// . . .
longjmp(envbuf, 5);
// Теперь setjmp будет исполнена заново,
// но вернет код, равный 5
```

См. также функцию `setjmp`, в описании которой приводится более подробный пример.

malloc

Прототип:

```
void *malloc(size_t size);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Выделяет блок памяти размером `size` байт и возвращает указатель на него. Этот способ поддерживается в основном в целях совместимости с С. Оператор С++ `new` выполняет те же действия, но имеет ряд преимуществ (см. также раздел «Ключевые слова — операторы» в главе «Операторы»). Тип `size_t` является целым, достаточно большим, чтобы содержать размер любого типа. В случае неудачи возвращаемым значением будет `NULL`.

В следующем примере показано выделение блока памяти для `n` целых чисел:

```
#include <stdlib.h>
int n;
int *p;
//..
p = reinterpret_cast<int*>(malloc(sizeof(int) * n));
if (p == NULL)
// Вывод сообщения об ошибке и завершение программы
```




ВНИМАНИЕ Нужно отметить необходимость приведения типа к требуемому виду (в примере — к `int*`); при использовании оператора `new` такой необходимости нет, так как она автоматически включена в действия оператора `new`.

См. также функции `free`, `calloc` и `realloc`.

modf

Прототип:

```
double modf(double x, double *intptr);
```

Файл заголовков: `math.h` или `cmath`.

Разбивает `x` на целую и дробную части. Та и другая имеют знак, совпадающий со знаком аргумента. Функция `modf` дробную часть числа возвращает, а целую часть числа помещает по адресу, определяемому указателем `intptr`.

В следующем примере в переменную `f` будет занесено значение 0.14159, а в переменную `d` — значение 3:

```
#include <math.h>
//..
double d;
double f = modf(3.14159, &d);
```

См. также функции `ceil` и `floor`.

perror

Прототип:

```
void perror(const char *string);
```

Файл заголовков: `stdio.h` или `cstdio`.

Выводит сообщение об ошибке вида: «строка: сообщение», в котором «строка» определяется параметром `string`, а «сообщение» — кодом ошибки (код ошибки берется из глобальной переменной `errno`). Сообщение об ошибке заканчивается символом новой строки и выводится в стандартный поток сообщений об ошибках (`stderr`), который

обычно назначен на монитор. Аргумент `string` является префиксом всего сообщения.

Пример:

```
#include <stdio.h>
//..
errno = 1;
perror("ERROR MESSAGE");
```

Версия 6.0 Microsoft Visual C++ выведет следующее: ERROR MESSAGE: Operation not permitted.

pow

Прототип:

```
double pow(double x, double y);
```

Файл заголовков: `math.h` или `cmath`.

Вычисляет значение числа x в степени y .

В следующем примере выводятся значения квадрата, куба и квадратного корня числа x :

```
#include <math.h>
#include <iostream.h>
double x;
//..
cout<<"x в степени 2 = "<<pow(x,2)<<"\n";
cout<<"x в степени 3 = "<<pow(x,3)<<"\n";
cout<<"x в степени 0.5 = " << pow(x,0.5) << " \n";
```

printf

Прототип:

```
int printf(const char *format,...);
```

Файл заголовков: `stdio.h` или `cstdio`.

Выводит на стандартное устройство вывода значения переменных, перечисленных в списке, обозначенном многоточием (...), в формате, определенном строкой `format`. Символы, не являющиеся символами формата, выводятся без

каких-либо преобразований. Возвращает число реально выведенных символов.

Эта функция является стандартной для вывода данных в библиотеке языка C. Несмотря на то что в C++ объект потока `cout` также может использоваться для этих целей, функции `printf` вполне достаточно для вывода любых значений простейших типов. В главе «Ввод и вывод» сравниваются функция `printf` и объект `cout`.

Строка форматирования `format` содержит ноль или более полей, называемых спецификаторами формата; многоточие (...) указывает на то, что после параметра `format` могут присутствовать (или отсутствовать) другие параметры. Их наличие определяется спецификаторами формата: для каждого спецификатора должен быть свой аргумент соответствующего типа, причем порядок их расположения соответствует порядку расположения спецификаторов.

Рассмотрим пример вывода значений двух аргументов: целого `i` и вещественного `x` (им соответствуют спецификаторы формата `%d` и `%f`):

```
printf("i = %d and x = %f\n", i, x);
```

Спецификаторы формата

В табл. 10 приведены спецификаторы формата для функции `printf`.

Таблица 10. Спецификаторы формата функции `printf`

Спецификатор	Описание
<code>%c</code>	Вывод одного байта как символа кода ASCII
<code>%d</code> , <code>%i</code>	Вывод целого числа в десятичной системе счисления
<code>%u</code>	Вывод беззнакового целого числа в десятичной системе счисления
<code>%o</code>	Вывод целого числа в восьмеричной системе счисления

Спецификатор	Описание
%x, %X	Вывод целого числа в шестнадцатеричной системе счисления
%e	Вывод вещественного числа в «научном» формате, например 1.273110e+01
%E	Аналогично %e, но символ «e» заменяется на «E»
%f	Вывод вещественного числа в обычном формате
%g	Вывод вещественного числа в формате либо %e, либо %f в зависимости от того, какой из них более короткий
%G	Аналогично спецификатору %g, но символ «e» заменяется на «E»
%s	Вывод строки текста
%p	Вывод указателя в шестнадцатеричном формате (стандарт ANSI не описывает этого спецификатора, но он существует практически во всех реализациях, так как удобен для разработчиков)
%%	Вывод символа процента (%)

Модификаторы формата

Управление шириной поля, предоставляемого для размещения значения, выполняется указанием одного или двух чисел, задающих минимальное количество символов для размещения и точность (если ширины поля не хватает для размещения значения, автоматически выделяется большее количество позиций):

■ %-minC или %minC;

■ %-min.precisionC или %min.precisionC.

Здесь C — спецификатор формата из табл. 10. Значение десятичного числа min — минимальная ширина поля. Смысл

модификатора `precision`, также десятичного числа, зависит от спецификатора формата, с которым он используется:

- при выводе строки (спецификатор `%s`) `precision` указывает максимальное число символов для вывода;
- при выводе вещественного числа (спецификаторы `%f` или `%e`) `precision` указывает число цифр после десятичной точки;
- если выводится целое число (спецификаторы `%d` или `%i`), `precision` указывает минимальное число выводимых цифр, что приводит к выводу ведущих (начальных) нулей, если число представляется меньшим числом цифр, чем указано в `precision`.

Символ минус (-) указывает на то, что значение выравнивается по левому краю и, если нужно, дополняется пробелами справа. При отсутствии минуса значение выравнивается по правому краю и дополняется пробелами слева.

В следующем примере выводится строка текста и вещественное число:

```
#include <stdio.h>
//..
char *state = "Петербург";
double temp = 18.7;
printf("Температура в г. %s %5.4f.\n", state, temp);
```

В этом случае будет выведено Температура в г. Петербурге 18.7000.

putc

Прототип:

```
int putc(int ch, FILE *stream);
```

Файл заголовков: `stdio.h` или `cstdio`.

Выводит символ, размещенный в младшем байте аргумента `ch`, в поток указанного файла. В случае обнаружения ошибки функция возвращает значение `E0F`, в противном случае возвращается выведенный символ.

Функция `putc` идентична функции `fputc`. Отличие заключается в том, что `putc` часто реализуется в виде макроса, а не функции, что приводит к сокращению времени исполнения. Таким образом, использование функции `putc` более предпочтительно (см. раздел «Функции файлового ввода/вывода»).

putchar

Прототип:

```
int putchar(int ch);
```

Файл заголовков: `stdio.h` или `cstdio`.

Выводит символ, размещенный в младшем байте аргумента `ch`, в стандартный поток вывода. В случае обнаружения ошибки функция возвращает значение `E0F`, в противном случае возвращается выведенный символ. Более подробно функции консольного ввода/вывода рассмотрены в разделе «Функции консольного ввода/вывода».

puts

Прототип:

```
int puts(char *string);
```

Файл заголовков: `stdio.h` или `cstdio`.

Выводит строку в стандартный поток вывода с добавлением символа новой строки. В случае обнаружения ошибки функция возвращает значение `E0F`, в противном случае возвращается неотрицательное число (см. раздел «Функции консольного ввода/вывода»).

qsort

Прототип:

```
void qsort(void *buf, size_t num, size_t size, int  
(compare)(const void *, const void *));
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Выполняет сортировку массивов любых размеров и типов. Три аргумента, `buf`, `num` и `size`, описывают сортируемый

массив: `buf` — указатель на массив, `n` — число элементов, `size` — размер каждого элемента. Аргумент `compare` — адрес функции обратного вызова, которая должна сравнить два значения, переданных ей функцией `qsort`, и вернуть соответствующий код: меньше нуля (если первый аргумент меньше второго), нуль (если аргументы равны) или больше нуля (если первый аргумент больше второго). Аргументы, передаваемые ей, определены как `void*`, то есть внутри функции они должны приводиться к нужному типу.

Рассмотрим пример сортировки массива целых чисел. Функция `cmp` получает указатели типа `int*`. Так как в `cmp` их тип определен как `void*`, необходимо явное приведение:

```
#include <stdlib.h>
#include <iostream.h>
int cmp(const void *p1, const void *p2)
{
    int i1 = *(int *)p1;
    int i2 = *(int *)p2;
    return i1 < i2 ? -1: ( i1 == i2 ? 0: 1 );
}
void main()
{
    int dat[] = { 1, 100, 4, 81, 9, 64, 16, 49, 25, 36 };
    qsort(dat, sizeof(dat)/sizeof(int), sizeof(int),
    &cmp);
    for (int i=0; i<sizeof(dat)/sizeof(int); i++)
        cout << dat[i] << '\n';
}
```



ПРИМЕЧАНИЕ При сортировке строк нужно помнить, что они являются указателями типа `char*`, и их содержимое будет определяться как `char**`.

Функция сравнения будет выглядеть следующим образом:

```
#include <stdlib.h>
int cmp(const void *p1, const void *p2)
{
    char *s1 = *(char **)p1;
```



```
char *s2 = *(char **)p2;  
return strcmp(s1, s2);  
}
```

raise

Прототип:

```
int raise(int signal);
```

Файл заголовков: `signal.h` или `csignal`.

Посылает сигнал программе, идентифицируя ошибку во время ее выполнения. Эта функция устарела и оставлена только для совместимости с C, так как в C++ для таких целей используется механизм исключений. Возвращает нулевое значение в случае успеха и ненулевое — в противном случае. В стандартной библиотеке C/C++ определено несколько имен для значений сигналов, в конкретных реализациях их может быть существенно больше:

- SIGABRT — выполнение программы прекращено;
- SIGFPE — ошибка операции с плавающей точкой;
- SIGILL — недопустимая команда;
- SIGINT — прерывание пользователя по Ctrl+C;
- SIGSEGV — обращение по недопустимому адресу памяти;
- SIGTERM — нормальное завершение программы.

Пример использования функции `raise`:

```
#include <signal.h>  
//..  
raise(SIGFPE);
```

Генерация сигнала функцией `raise` обычно приводит к вызову функции реакции на сигнал (см. функцию `signal`).

rand

Прототип:

```
int rand(void);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Возвращает очередное псевдослучайное число в диапазоне от 0 до RAND_MAX (возвращаемое число является псевдослучайным, так как каждое следующее получается в результате сложных преобразований предыдущего). Для увеличения «случайности» получаемых чисел можно использовать изменение начального значения функцией srand. Пример генерации чисел:

```
#include <stdlib.h>
#include <time.h>
#include <iostream.h>
//..
srand(time(NULL)); rand();
for (int i = 0; i < 10; i++)
    cout << rand();
```

Делением полученного псевдослучайного числа на RAND_MAX получается псевдослучайное число в диапазоне от 0.0 до 1.0, далее легко получить псевдослучайные числа в любом диапазоне:

```
#include <stdlib.h>
// Генерация псевдослучайных в диапазоне от 0 до 5
cout << floor(5 * (double)rand()/RAND_MAX) << endl;
```

realloc

Прототип:

```
void *realloc(void *memblock, size_t size);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Получает блок памяти (выделенный ранее одной из функций `malloc`, `calloc` или `realloc`), при необходимости перемещает его так, чтобы он соответствовал новому размеру. При этом ранее размещенные в нем данные не теряются. Большинство правил, относящихся к `malloc`, действительны и для `realloc`.

Значение, возвращаемое функцией, является указателем на новое положение блока памяти. Оно отличается от старого значения (`memblock`) только в том случае, если `realloc`

пришлось переместить блок памяти. Если не удалось увеличить блок памяти, то возвращаемое значение равно NULL, причем значение `memblock` остается доступным программе.

В следующем примере сначала выделяется блок размером `n` байт, а позднее его размер увеличивается в два раза:

```
#include <stdlib.h>
int n;
char *p;
//..
p = reinterpret_cast<char*>(malloc(n));
if (p == NULL)
    // Недостаточно памяти. Сообщение об ошибке
//..
p = reinterpret_cast<char*>(realloc(p, 2 * n));
if (p == NULL)
    // Недостаточно памяти. Сообщение об ошибке
```

См. также функции `malloc`, `calloc` и `free`.

remove

Прототип:

```
int remove(const char *filename);
```

Файл заголовков: `stdio.h` или `cstdio`.

Удаляет существующий файл. В случае успеха возвращает нулевое значение, иначе — ненулевое.

rename

Прототип:

```
int rename(const char *oldname, const char *newname);
```

Файл заголовков: `stdio.h` или `cstdio`.

Переименовывает существующий файл или папку. В случае успеха возвращает нулевое значение, иначе — ненулевое. Имя `oldname` должно указывать на существующий файл или папку. Если `oldname` является именем файла, то можно переместить этот файл в другую папку.

Пример переименования файла и перемещения файла:

```
#include <stdio.h>
//...
rename("junk.txt", "useful.txt");
rename("c:\\memo.txt", "c:\\mydocs\\memo.txt");
```

scanf

Прототип:

```
int scanf(const char *format, ...);
```

Файл заголовков: `stdio.h` или `cstdio`.

Вводит данные (строки, целые числа, числа с плавающей точкой и т. д.) со стандартного устройства ввода в формате, определяемом строкой `format`. Возвращает число реально введенных значений.

Перед вводом `scanf` никакого приглашения не выводит, его можно организовать функцией `printf`. Несмотря на то что для этих целей в C++ также может использоваться объект потока `cin`, функции `scanf` вполне достаточно для ввода любых значений простейших типов C и C++.

При вызове функции должен быть указан, по крайней мере, один параметр, строка формата с возможными дополнительными параметрами, соответствующими переменным, вводимым согласно формату. При этом должны указываться адреса, а не сами переменные. Символы в строке `format`, не являющиеся символами формата, игнорируются.

Строка формата `string` содержит ноль или более полей, называемых спецификаторами формата. Для каждого спецификатора должен быть свой адрес аргумента соответствующего типа, причем порядок их расположения соответствует порядку расположения спецификаторов формата. Рассмотрим пример ввода значения целого `i`:

```
scanf("%d", &i);
```

Спецификаторы формата

В табл. 11 приведен список спецификаторов формата функции `scanf`.

Таблица 11. Спецификаторы формата функции `scanf`

Спецификатор	Описание
<code>%c</code>	Ввод одного символа, даже если это пробел
<code>%d</code>	Ввод целого числа как десятичного
<code>%I</code>	Ввод целого числа как десятичного, если число начинается символом нуля (0), то оно вводится как восьмеричное, если число начинается символами 0x или 0X, то оно вводится как шестнадцатеричное
<code>%u</code>	Ввод беззнакового целого числа как десятичного
<code>%o</code>	Ввод целого числа как восьмеричного
<code>%x, %X</code>	Ввод целого числа как шестнадцатеричного
<code>%e, %f, %g</code>	Ввод вещественного числа
<code>%s</code>	Ввод строки текста до ближайшего пробела, табуляции или конца строки
<code>%p</code>	Ввод указателя в шестнадцатеричном формате
<code>%[символы]</code>	Ожидается символ из указанного списка (не вводится)
<code>%%</code>	Ожидается символ процента (не вводится)

Все аргументы функции `scanf`, указанные следом за аргументом `format`, должны быть адресами. Для обычных переменных используется оператор определения адреса (`&`). Но имена строк сами являются адресами (массивов символов), поэтому оператор определения адреса к ним применять не нужно:


```
#include <stdio.h>
//..
char name[256];
int id;
printf("Введите имя и номер: ");
scanf("%s %i", name, &id);
```

Модификаторы формата

Между символом процента (%) и спецификатором формата могут появляться символы-модификаторы. Модификатор `l` при вводе вещественного числа означает, что вводится число типа `double`, а не `float`:

```
double x, y, z;
float f;
scanf("%lf %lf %lf %f", &x, &y, &z, &f);
```

Аналогично модификатор `h` используется для ввода коротких целых, и модификатор `l` — для ввода длинных целых.

Существует еще один тип модификатора — звездочка (*). Его использование указывает, что соответствующее значение нужно прочесть, но никуда не помещать («потерять»).

Другие символы в строке формата

Строка формата `string` функции `scanf` может содержать символы, отличные от спецификаторов формата, в этом случае `scanf` ожидает встретить эти символы при вводе, но никуда их не помещает.

Наличие пробелов и/или символов табуляции в строке формата указывает, что между вводимыми значениями ожидаются эти разделители (в произвольном количестве).

Если `scanf`, согласно строке формата, ожидает некий символ, но его на текущей позиции не обнаруживается, то `scanf` завершает свою работу и возвращает количество введенных символов.

setjmp

Прототип:

```
int setjmp(jmp_buf envbuf);
```


Файл заголовков: `setjmp.h` или `csetjmp`.

Наряду с функцией `longjmp` предоставляет возможность перехода между функциями. Эта методика совершенно не согласуется со структурным программированием, и ее следует избегать (использование этих функций может оказаться полезным только при системном программировании).

Действие функции `setjmp` заключается в сохранении переменной окружения в переменной `envbuf`. Далее, при исполнении функции `longjmp`, окружение восстанавливается из `envbuf` (включая и адрес исполняемого кода, то есть управление передается в функцию `setjmp`). Функция `setjmp` изначально возвращает значение, равное 0. При повторном исполнении `setjmp` (как результат вызова `longjmp`) она возвращает значение аргумента `status` функции `longjmp`. Если оно не равно нулю, то таким способом можно отличить начальное исполнение `setjmp` от исполнения после вызова функции `longjmp`. Пример использования функций `setjmp` и `longjmp`:

```
#include <setjmp.h>
#include <iostream.h>
jmp_buf envbuf;
int status;
void test_func(void)
{
    cout << "Вот функция test_func." << endl;
    longjmp(envbuf, 5); // возврат в main
}
void main()
{
    status = setjmp(envbuf);
    cout << "После setjmp." << endl;
    if (status == 0)
        test_func();
    cout << "status = " << status << endl;
}
```

Программа сначала вызывает функцию `setjmp` для регистрации точки положения. Возвращаемое значение равно 0.

При выполнении функции `test_func` происходит обратный переход в ту же точку. Фактически происходит повторное исполнение функции `setjmp`, но на этот раз она завершается с ненулевым значением (5), которое используется для предотвращения повторного вызова функции `test_func`. Во время работы программы выводится следующий протокол:

После `setjmp`.

Вот функция `test_func`.

После `setjmp`.

`status = 5`

Отметим, что для успешного исполнения функции `longjmp` функция, которая вызывала `setjmp`, не должна быть функцией завершения (например, зарегистрированной функцией `atexit`). Аналогично, если функция, вызвавшая `setjmp`, изменяет значение своих локальных переменных, то значения становятся неопределенными после исполнения `longjmp` (исключением являются переменные, объявленные как `volatile`, с которыми эти проблемы не возникают).

setlocale

Прототип:

```
char *setlocale(int type, const char *locale);
```

Файл заголовков: `locale.h` или `clocale`.

Устанавливает новую локализацию программы, которая подразумевает набор символов, форматы представления чисел, денежных единиц и т. п., для конкретной страны/местности. Функция `setlocale` также может использоваться для получения текущих параметров локализации.

Параметр `type` указывает тип устанавливаемой категории локализации и может принимать следующие значения:

- `LC_ALL` — все категории;
- `LC_COLLATE` — набор символов, используемый функцией `strcoll`;
- `LC_CTYPE` — набор символов, распознаваемый функцией `isalnum`;

- LC_MONETARY — формат представления денежных сумм;
- LC_NUMERIC — формат представления чисел функциями типа printf;
- LC_TIME — формат представления даты/времени функциями типа asctime.

Аргумент locale является строкой, содержащей новые параметры локализации. Стандарт ANSI определяет только строки локализации "C" и "", что означает «локализацию C» и «локализацию по умолчанию». Другие локализации определяются конкретной версией компилятора. Если аргументом locale является NULL, то функция локализации не изменяет, а только возвращает текущую локализацию. Если аргумент locale содержит строку, отличную от NULL, то возвращаемым значением является строка новой локализации.

В следующем примере демонстрируется использование функции setlocale для установки числового формата:

```
#include <stdio.h>
#include <locale.h>
void main()
{
    puts(setlocale(LC_NUMERIC, "French") );
    printf("%f\n", 8.007);
}
```

При исполнении этой программы, откомпилированной Microsoft Visual C++, будет выведено следующее:

```
French_France.1252
8,0070000
```

Отметим, что в качестве разделителя целой и дробной частей числа используется запятая, а не точка.

signal

Прототип:

```
void (*signal(int signum, void (*func)(int)))(int);
```

Файл заголовков: signal.h или csignal.

Регистрирует обработчик сигналов. Несмотря на то что обработчики исключений C++ обеспечивают лучший механизм управления ошибками, функции `signal` и `raise` включены в библиотеку для совместимости с C.

Несмотря на довольно сложный вид определения функции, оно достаточно просто. Функция `signal` имеет два аргумента:

```
signal(signum, func)
```

Аргумент `signum` может быть одним из предопределенных значений `SIGINT`, `SIGILL` или `SIGFPE` (более полный список приведен в описании функции `raise`).

Аргумент `func` может быть одним из специальных значений: `SIG_DFL` (установка по умолчанию) или `SIG_IGN` (игнорирование сигнала). Например, следующий код устанавливает игнорирование сигналов `SIGINT`, `SIGILL` и `SIGTERM`:

```
signal(SIGINT, SIG_IGN);  
signal(SIGILL, SIG_IGN);  
signal(SIGFPE, SIG_IGN);
```

Также этот аргумент может содержать адрес специальной функции — обработчика сигналов (принимающей целое как аргумент и ничего не возвращающей). Например, следующий код регистрирует обработчик двух различных сигналов, а затем генерирует эти сигналы для проверки. Аргумент обработчика используется для определения типа сигнала:

```
#include <signal.h>  
#include <iostream.h>  
void handler(int sig)  
{  
    cout << "Signal received: " << sig << '\n';  
}  
//...  
signal(SIGINT, &handler);  
signal(SIGILL, &handler);  
raise(SIGINT);  
raise(SIGILL);
```


При выходе из обработчика сигнала управление автоматически передается на следующую за вызвавшей сигнал инструкцию. Исключением является сигнал SIGFPE, когда поведение на выходе является неопределенным.

Правила для написания обработчика сигналов

При написании обработчика сигналов необходимо соблюдать ряд правил. Существуют требования, накладываемые стандартом ANSI (перечислены ниже) и накладываемые конкретной реализацией компилятора (см. документацию компилятора). Общие требования стандарта ANSI:

- в случае сигнала SIGFPE необходимо организовать выход при помощи функций `setjmp` и `longjmp`;
- не использовать функции `setjmp` и `longjmp` для обработки сигналов, отличающихся от SIGFPE;
- избегать использования функций, определенных в файле `stdio.h` (типа `printf`), функций работы с блоками памяти (типа `malloc`) и функций получения системного времени;
- в Windows NT и Windows 95 (и более поздних) сигнал SIGINT не генерируется при нажатии клавиш Ctrl+C. Однако он может возбуждаться при помощи функции `raise`.

Возвращаемое значение

Функция `signal` возвращает адрес ранее установленного обработчика данного сигнала. Так, если функция `signal` используется для регистрации обработчика сигнала SIGINT, то возвращаемое значение содержит адрес предыдущего обработчика SIGINT, чем можно воспользоваться для временного изменения обработчика:

```
void (*saved_handler)(int);  
saved_handler = signal(SIGINT, &handler);  
//..  
signal(SIGINT, saved_handler);
```


sin

Прототип:

```
double sin(double x);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает значение синуса x ($\sin x$), получая значение угла в радианах. Возвращаемое значение находится в диапазоне от -1 до 1 .

В следующем примере печатается значение синуса $\pi/2$. Результат может не совпадать с тем, который приводится в учебниках по математике, но значение должно быть близко к 1 :

```
#include <math.h>
#include <iostream.h>
//..
double pi = atan(1.0) * 4;
cout << sin(pi/2) = " << sin(pi/2) << '\n';
```

См. также функции `acos`, `asin`, `atan`, `atan2`, `cos` и `tan`.

sinh

Прототип:

```
double sinh(double x);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает значение гиперболического синуса x , который является углом, заданным в радианах. Если значение функции окажется вне диапазона представимых чисел, то функция возвращает значение `HUGE_VAL`, а переменная `errno` получает значение `ERANGE`.

См. также функции `cosh` и `tanh`.

sprintf

Прототип:

```
int sprintf(char *buffer, const char *format,...);
```

Файл заголовков: `stdio.h` или `cstdio`.

Аналогично функции `printf` выводит отформатированные данные, но не на консоль, а в текстовую строку (которую впоследствии можно использовать для вывода на консоль).

Первый параметр функции (`buffer`) является адресом буфера, куда помещается отформатированная строка. Этот буфер должен быть достаточного размера, чтобы не произошло его переполнения. Как и в функции `printf`, многоточие указывает на возможные дополнительные аргументы при вызове функции. Пример использования `sprintf`:

```
#include <stdio.h>
//..
char buf[256];
double x = 2.50e3;
sprintf(buf, "x = %f", x);
puts(x);
```

Вывод этой программы:

```
x = 2500.000000
```

Подробнее о строке формата можно прочитать в описании функции `printf`.

sqrt

Прототип:

```
double sqrt(double x);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает значение квадратного корня x .

В следующем примере выводится значение квадратного корня из 2:

```
#include <math.h>
#include <iostream.h>
//..
cout << "Квадратный корень из 2 равен " << sqrt(2) << '\n';
```

srand

Прототип:

```
void srand(unsigned int seed);
```


Файл заголовков: `stdlib.h` или `cstdlib`.

Устанавливает начальное значение последовательности псевдослучайных чисел.

Эта функция предназначена для того, чтобы изменять последовательности псевдослучайных чисел, так как если не изменять начального значения (`seed`), то при запусках программы будут получаться одинаковые последовательности псевдослучайных чисел. Простейшее решение — использовать текущее системное время в качестве начального значения:

```
#include <stdlib.h>
#include <time.h>
//..
srand(time(NULL));
rand();
```

См. также функцию `rand`.

sscanf

Прототип:

```
int sscanf(const char *buf, const char *format,...);
```

Файл заголовков: `stdio.h` или `cstdio`.

Аналогично функции `scanf` вводит данные, но не с клавиатуры, а из строки символов, переданной ей первым параметром. Подобные действия можно выполнить с помощью других функций типа `strtod` или `atof` (для получения вещественного числа), `atoi` (для получения целого числа) и т. п. Однако функция `sscanf` позволяет совместить возможности перечисленных функций с возможностью ввода нескольких значений из одной строки.

Аргумент `buf` — строка символов, из которой вводятся значения, `format` — строка формата, в соответствии с которой происходит преобразование данных, а многоточие указывает на наличие необязательных аргументов, соответствующих адресам вводимых значений:


```
#include <stdio.h>
//..
char data[] = "1.234";
double x;
sscanf(data, "%lf", &x);
```

Список спецификаторов формата и их смысл можно посмотреть в описании функции `scanf`.

strftime

Прототип:

```
size_t *strftime(char *str, size_t max_size, const char *fmt,
const struct tm *time);
```

Файл заголовков: `time.h` или `ctime`.

Формирует строку `str`, содержащую дату и время в соответствии со строкой формата `fmt`. Параметр `time` — указатель на структуру `tm`, в которой содержится информация о дате и времени. В `fmt` могут содержаться как спецификаторы формата, так и другие символы, которые попадут в выходную строку без преобразования. Аргумент `max_size` указывает максимальное число формируемых символов в строке `str`. Возвращаемое значение функции — число сформированных символов (длина `str`).

Спецификаторы формата

Спецификаторы, которые могут содержаться в строке формата, перечислены в табл. 12.

Таблица 12. Спецификаторы формата `strftime`

Спецификатор	Описание
%a	Аббревиатура названия дня недели
%A	Название дня недели
%b	Аббревиатура названия месяца
%B	Название месяца
%c	Стандартная строка даты и времени

продолжение ↗

Таблица 12. Спецификаторы формата strftime (продолжение)

Спецификатор	Описание
%d	День месяца (1–31)
%H	Час суток (0–23)
%I	Час суток (1–12)
%j	День года (1–366)
%m	Номер месяца (1–12)
%M	Минуты (0–59)
%p	Указатель до/после полудня: AM/PM
%S	Секунды (0–61): да — может быть до 61!
%U	Номер недели в году: воскресенье — первый день (0–53)
%w	День недели: воскресенье — первый день (0–6)
%W	Номер недели в году: понедельник — первый день (0–53)
%x	Дата с учетом локализации
%X	Дата с учетом локализации
%y	Две последних цифры года
%Y	Номер года (4 цифры)
%Z	Название временной зоны
%%	Символ процента

В следующем примере выводится текущая дата и время в нескольких различных форматах:

```
#include <stdio.h>
#include <time.h>
void main()
```



```
{
    char s[100];
    time_t t = time(NULL);
    struct tm *tmp = localtime(&t);
    strftime(s, 99, "The month is %B.", tmp);
    puts(s);
    strftime(s, 99, "The day of the week is %A.", tmp);
    puts(s);
    strftime(s, 99, "The time is %H:%M %p.", tmp);
    puts(s);
}
```

Эта программа выводит:

The month is March.

The day of the week is Sunday.

The time is 12:59 PM.

strtod

Прототип:

```
double strtod(const char *str, char **end);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Преобразует строку символов в числовое значение и возвращает его. За небольшими исключениями действует аналогично функции `atof`. Функция пропускает ведущие пробелы и преобразовывает последующие символы до тех пор, пока они допустимы в записи вещественного числа. Например, «-24.56ху7» преобразуется в -24.56. Функция воспринимает символы «Е» и «е» как начало экспоненциальной части, так что «5Е2» преобразуется в 500.

В отличие от функции `atof` функция `strtod` устанавливает указатель на позицию первого символа, который не интерпретируется как часть вещественного числа. Также в случае переполнения (то есть попытки преобразования слишком большого числа) `strtod` возвращает значение `HUGE_VAL` или `-HUGE_VAL` (в зависимости от знака в строке). Аналогично в случае антипереполнения (числа, очень близкого

к нулю, но не равного ему, которое не может быть представлено как вещественное, например: 5E-999) возвращает значение 0.0. В обоих случаях также устанавливается значение ERANGE в глобальной переменной `errno`. Рассмотрим пример чтения значения с клавиатуры:

```
#include <stdlib.h>
#include <stdio.h>
//..
char *p;
char str[100];
double x = strtod(gets(str), &p);
```

См. также функции `atoi`, `atol`, `atof`, `strtol` и `strtoul`.

strtol

Прототип:

```
double strtol(const char *str, char **end, int radix);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Преобразует строку символов в числовое значение с учетом указанной системы счисления `radix` и возвращает полученное число. Например, значение `radix`, равное 16, означает преобразование шестнадцатеричного значения. Функция пропускает возможные начальные пробелы и заканчивает преобразование на первом символе, который не может появиться в образе числа. Параметр `end` является адресом указателя типа `char*`; этот указатель будет содержать адрес первого непреобразованного символа.

В следующем примере читается строка FF как шестнадцатеричное число, которое в десятичной системе является числом 255:

```
#include <stdlib.h>
//..
char *p;
long n = strtol("FF", &p, 16);
```

См. также функции `atoi`, `atol`, `atof`, `strtod` и `strtoul`.

strtoul

Прототип:

```
double strtoul(const char *str, char **end, int radix);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Преобразует строку символов в числовое значение с учетом системы счисления. Эта функция аналогична функции `strtol`, но, в отличие от нее, выполняет преобразование в беззнаковое длинное целое (и, соответственно, не воспринимает знака числа).

В следующем примере читается строка FF как шестнадцатеричное число, которое в десятичной системе является числом 255:

```
#include <stdlib.h>
//..
char *p;
unsigned long n = strtoul("FF", &p, 16);
```

См. также функции `atoi`, `atol`, `atof`, `strtod` и `strtol`.

system

Прототип:

```
int system(const char *command);
```

Файл заголовков: `stdlib.h` или `cstdlib`.

Передаёт строку командному процессору операционной системы. Это позволяет исполнить другую программу или утилиту. Возвращаемое значение зависит от реализации, но, как правило, значение ноль (0) указывает на успешное завершение.

Пример использования функции `system` для вывода списка папок и файлов (пример для DOS или Windows):

```
#include <stdlib.h>
// или #include <cstdlib>
//..
system("DIR /W");
```


tan

Прототип:

```
double tan(double x);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает значение тангенса x ($\operatorname{tg} x$), где x задается в радианах, и возвращает отношение двух сторон треугольника. В следующем примере печатается значение тангенса $\pi/4$, равное 1:

```
#include <math.h>
#include <iostream.h>
//...
double pi = atan(1) * 4;
cout << "tan(pi/4)=" << tan(pi/4) << '\n';
```

См. также функции `acos`, `asin`, `atan`, `atan2`, `cos` и `sin`.

tanh

Прототип:

```
double tanh(double x);
```

Файл заголовков: `math.h` или `cmath`.

Возвращает значение гиперболического тангенса x .

См. также функции `sinh` и `cosh`.

tmpfile

Прототип:

```
FILE *tmpfile(void);
```

Файл заголовков: `stdio.h` или `cstdio`.

Открывает временный файл в режиме двоичного ввода/вывода. Этот файл будет автоматически удален при закрытии файла или завершении программы. Как и функция `fopen`, функция `tmpfile` возвращает указатель на открытый поток или `NULL` (в случае ошибки). Удобство этой функции в том, что можно организовать запись большого массива данных, затем повторно их использовать и не

заботиться об удалении такого файла по завершении задачи, что особенно удобно в случае аварийного завершения программы.

tolower

Прототип:

```
int tolower(int ch);
```

Файл заголовков: `ctype.h` или `sctype`.

Получает параметр `ch` и возвращает его в нижнем регистре (если он является символом буквы верхнего регистра) или сам символ. В параметре `ch` используется только младший байт.

Пример перевода строки в строку, содержащую все буквы в нижнем регистре:

```
#include <string.h>
#include <ctype.h>
void convert_to_lower(char s[])
{
    int i;
    for (i = 0; i < strlen(s); i++)
    {
        s[i] = tolower(s[i]);
    }
}
```

После обработки строки: `I used to work the FBI.` приведенной выше функцией имеем на выходе строку: `i used to work the fbi.`

toupper

Прототип:

```
int toupper(int ch);
```

Файл заголовков: `ctype.h` или `sctype`.

Получает параметр `ch` и возвращает его в верхнем регистре (если он является символом буквы нижнего регистра)

или сам символ. В параметре `ch` используется только младший байт.

Пример перевода строки в строку, содержащую все буквы в верхнем регистре:

```
#include <string.h>
#include <ctype.h>
void convert_to_upper(char s[])
{
    int i;
    for (i = 0; i < strlen(s); i++)
        {s[i] = toupper(s[i]);}
}
```

После обработки строки: `I used to work the FBI.` приведенной выше функцией имеем на выходе строку: `I USED TO WORK THE FBI.`

Объекты и классы библиотеки ввода/вывода

В главе «Ввод и вывод» вводятся стандартные объекты C++ `cin` и `cout`, обеспечивающие ввод/вывод в C++. Однако C++ предлагает значительно большие возможности, в том числе большее разнообразие форматов ввода/вывода для чтения из файла и записи в файл.

В ANSI C++ введены новые классы библиотек для поддержки объектов `cin`, `cout` и других связанных с ними объектов. Они могут использоваться так же, как и классы стандартных библиотек C++, но с некоторыми отличиями. Для простоты в этой главе основное внимание уделено традиционным классам C++, а там где это нужно описываются основные различия между старым и новым стилями программирования ввода/вывода.

Краткий обзор библиотеки классов ввода/вывода

Библиотека классов ввода/вывода в C++ обеспечивает некоторое преимущество над функциями `printf` и `scanf`, но в обмен на эти преимущества приходится сталкиваться с понятием классов и их иерархии.

На самом простом уровне для чтения и записи информации используются объекты `cin` и `cout`. Так, например, программа, приведенная ниже, выводит поясняющее сообщение, а затем считывает значение переменной.

```
#include <iostream.h>
void main()
{
    cout << "Введите значение переменной x:";
    cin >> x;
    return 0;
}
```


Попробуем расширить класс `MyClass` так, чтобы он мог отправлять и получать вводимую информацию. Стандартные объекты `cin` и `cout` — это объекты классов `istream` и `ostream` соответственно. Если правильно определить функции оператора, то расширенный класс `MyClass` будет работать со стандартными классами ввода/вывода, и для ввода объекта можно будет использовать инструкции такого вида:

```
MyClass obj;  
cin >> obj;  
//Объект инициализируется с клавиатуры
```

В следующем разделе «Организация ввода/вывода с помощью классов» описана функция, которая позволяет работать с данной программой.

Иногда нужно, помимо обычного чтения или записи информации, сделать это в определенном формате, например, в шестнадцатеричном или восьмеричном. Для этого используются манипуляторы ввода/вывода `hex` и `oct` соответственно. Следующая программа выводит на экран значение переменной `n` в восьмеричной системе счисления и затем переводит курсор на следующую строку с помощью манипулятора `endl`:

```
#include <iostream.h>  
void main()  
{  
    int n=123;  
    cout << oct << n << endl;  
}
```

Программа выводит на экран монитора число 123 в восьмеричной системе счисления — 173.

Подробнее все манипуляторы ввода/вывода, в том числе `hex`, `oct` и `endl`, описываются в разделе «Манипуляторы ввода/вывода».

Для более полного контроля можно установить флаги форматирования с помощью вызова функции `setf`, члена базового класса `ios`. Класс `ios` определяет также несколько битовых флагов, занимающих один бит и принимающих

значения ноль или единица. Кроме `setf`, классы ввода/вывода поддерживают целый ряд функций-членов. Их описания можно найти в разделе «Обзор объектов и классов ввода/вывода в C++». Этот раздел обобщает классы `istream` и `ostream` и связанные с ними классы:

- `fstream`, класс ввода/вывода для файлов;
- `ifstream`, класс чтения из файла;
- `istrstream`, класс чтения из строки;
- `ofstream`, класс записи в файл;
- `ostrstream`, класс записи в строку;
- `strstream`, класс ввода/вывода для строк.

Организация ввода/вывода с помощью классов

Самое большое преимущество классов ввода/вывода по сравнению с функциями из файла `stdio.h` — их расширяемость. Говоря простым языком, можно переопределить операторы ввода/вывода потоков для работы с объектами класса, созданного пользователем. Таким образом, можно полностью контролировать формат ввода и вывода информации для собственных классов.

Переопределить функции типа `printf` для новых классов невозможно.

Расширение оператора вывода

Для работы с объектами `cout` и `cerr` необходимо переопределить оператор вывода `<<`. Смысл данного оператора состоит в том, что он определяет способ взаимодействия между классом `ostream`, объектом которого является `cout`, и собственным классом (классом, в котором он определен). Для функции этого оператора нужно написать новый прототип, который включается в описание класса:

```
class MyClass
{
    // . . .
    friend ostream& operator<<(ostream&, MyClass&);
```



```

    //Объявление вводимого оператора
    //. . .
};

```

Имя класса `MyClass` выбрано произвольно, и вместо него можно использовать любое другое. В связи с тем, что левый операнд не является объектом класса `MyClass`, необходимо объявить функцию с ключевым словом `friend` (см. также раздел «Для чего нужны функции-друзья» в главе «Перегрузка операторов»).

Введем теперь определение объявленной функции:

```

ostream& operator<<(ostream& arg1, MyClass&, arg2)
{
    //. . . инструкции
    return arg1;
}

```



ПРИМЕЧАНИЕ Функция должна возвращать указатель на объект класса `ostream`, тип которого совпадает с типом левого операнда, чтобы обеспечить возможность использования нескольких последовательных операторов вывода в одной инструкции.

Далее следует пример создания класса, основанного на использовании класса `ostream`.

```

#include <iostream.h>
class MyClass
{
    long x,y;
    public:
        //конструктор
        MyClass(long newx, long newy) {x=newx; y=newy;}
        friend ostream& operator<<(ostream&, MyClass&);
    //. . .
};
ostream& operator<<(ostream& fo, MyClass& fp)
{
    fo << fp.x << " " << fp.y;
    return fo;
}

```


Определив класс таким образом, можно в основной программе использовать следующий оператор вывода:

```
MyClass pr(75, 1);  
cout << pr <<endl;
```

При этом через пробел будут выведены значения полей объекта `pr`, то есть строка:

```
75 1
```

Расширение оператора ввода

Точно так же, как в классе `ostream` перегружается функция оператора сдвига влево (`<<`), в классе `istream` перегружается функция оператора сдвига вправо (`>>`). Для перегрузки функции оператора сдвига вправо для класса `MyClass` нужно объявить функцию:

```
friend istream& operator>>(istream&, MyClass&);
```

Вместо имени `MyClass` можно использовать любое другое.

Функция определяется таким образом:

```
istream& operator>>(istream& arg1, MyClass& arg2)  
{  
    // . . . инструкции  
    return arg1;  
}
```

Вместо параметров `arg1` и `arg2` можно использовать любые другие имена. Рассмотрим пример.

```
#include <iostream.h>  
class MyClass  
{  
    long x,y;  
    public:  
        MyClass(long newx, long newy) {x=newx; y=newy;}  
        friend ostream& operator<<(ostream&, MyClass&);  
        friend istream& operator>>(istream&, MyClass&);  
};  
ostream& operator<<(ostream& fo, MyClass& fp)
```



```
{
    fo << fp.x << " " << fp.y;
    return fo;
}
istream& operator>>(istream& fi, MyClass& fp)
{
    fi >> fp.x >> fp.y;
    return fi;
}
```

Данная программа является примером использования обоих классов: `ostream` и `istream`.



ВНИМАНИЕ Функции операторов `<<` и `>>` должны быть совместимы, чтобы выведенную информацию об объекте можно было использовать для его ввода. Это важно, так как классы `istream` и `ostream` наследуются классами файлового ввода/вывода `ifstream` и `ofstream`. Приведенный пример обеспечивает такую совместимость.

С помощью инструкции `fo << fp.x << " " << fp.y;` значения выводимых переменных `fp.x` и `fp.y` разделяются пробелом. Класс `istream` в дальнейшем обеспечивает возможность правильного считывания значений переменных, записанных таким образом. При разделении значений другими способами, например, с помощью запятой, могут возникнуть проблемы.



ПРИМЕЧАНИЕ Чтение и запись текстовых переменных сопряжены с некоторыми трудностями. Дело в том, что переменные надо разделять каким-то образом. Если использовать для считывания значения текстовой переменной оператор `>>`, определенный в предыдущем примере, то символы будут читаться только до пробела. Простое решение этой проблемы — запись каждой переменной в новую строку (см. также описание функции `getline` в разделе «`istream`»).

Манипуляторы ввода/вывода

Манипуляторы ввода/вывода — это определенные в C++ объекты, которые воздействуют на информацию, изме-

няя ее формат или добавляя к ней управляющие символы (табл. 13).

Таблица 13. Манипуляторы ввода/вывода

Манипуляторы ввода/вывода	Описание
dec	Определяет десятичный формат вывода
endl	Добавляет символ новой строки
ends	Добавляет нуль-терминатор
flush	Пересылает информацию, находящуюся в буфере, на физический диск
hex	Определяет шестнадцатеричный формат вывода
oct	Определяет восьмеричный формат вывода

Ниже приведена программа, которая выводит значение переменной `h` как шестнадцатеричное и значение переменной `o` как восьмеричное.

```
#include <iostream.h>
void main()
{
    int h,o;
    cout << hex << h << " " << o << endl;
}
```



ПРИМЕЧАНИЕ Напомним, что оператор сдвига влево (`<<`) выполняется слева направо и возвращает ссылку на левый операнд, то есть снова на `cout`.

Таким образом, последняя инструкция примера выполняется в следующей последовательности:


```
cout << hex;      /*переход к шестнадцатеричному формату*/  
cout << h;        //вывод h  
cout << " ";      //вывод пробела  
cout << oct;      /* переход к восьмеричному формату*/  
cout << o;        //вывод o  
cout << endl;     //переход на новую строку
```

Точно таким же образом манипуляторы ввода/вывода работают и с объектами чтения:

```
cin >> hex >> h >> oct >> o;
```

Флаги форматирования

В языке C++ каждый объект класса `ios` обладает внутренними флагами форматирования, каждый из которых представляет собой бит, имеющий определенное значение. Эти флаги могут быть установлены или сняты с помощью функций `setf` и `unsetf` класса `ios`. В табл. 14 описаны флаги форматирования ввода/вывода.

Таблица 14. Флаги форматирования ввода/вывода

Флаг	Описание
dec	Использует десятичную систему счисления для целых чисел
fixed	Показывает вещественные числа в стандартной системе обозначений, то есть с десятичной точкой (например 503.5)
hex	Использует шестнадцатеричную систему счисления для целых чисел
internal	Заполняет символами пространство между знаком или основанием системы счисления и данными, если ширина поля больше, чем размер поля вывода
left	Выравнивает символы по левой границе, если ширина поля больше, чем размер поля вывода

Флаг	Описание
oct	Использует восьмеричную систему счисления для целых чисел
right	Выравнивает символы по правой границе, если ширина поля больше, чем размер поля вывода
scientific	Показывает числа с плавающей запятой в научном формате (например 5.035e2)
showbase	Выводит 0X или 0 перед шестнадцатеричными или восьмеричными значениями соответственно
showpoint	Отображает все вещественные числа с обязательной десятичной точкой и конечными нулями
showpos	Показывает знак +, если число положительное
skipws	Пропускает начальные пробелы при вводе
stdio	Записывает содержимое потоков stdout и stderr на физический диск после каждой операции вывода
unitbuf	Записывает содержимое буфера на физический диск после каждой операции вывода
uppercase	Выводит в верхнем регистре шестнадцатеричные цифры, символ экспоненты E и шестнадцатеричный префикс 0X

Все флаги являются членами класса `ios`. Чтобы установить какой-либо флаг, необходимо вызвать функцию `setf` любого объекта потоков (например `cout`) и флаг с префиксом `ios::`.

```
#include <iostream.h>
void main()
```



```
{  
    cout.setf(ios::scientific);  
    cout.setf(ios::showpos);  
    cout << 503.5 << endl;  
}
```

В связи с тем что флаги являются отдельными битами, можно применять поразрядный оператор ИЛИ (|) для одновременной установки нескольких флагов:

```
cout.setf(ios::scientific | ios::showpos);
```

Флаги можно не только устанавливать, используя функцию `setf`, но и снимать с помощью функции `unsetf`:

```
cout.unsetf(ios::showpos);
```

Каждый потоковый класс и объект поддерживает три функции: `setf`, `unsetf`, `flags`. Функция `flags` может быть использована для получения текущей установки флагов или для выполнения одновременной установки всех флагов.

Необходимо помнить, что формат ввода/вывода для каждого потока остается неизменными до тех пор, пока его установки не изменены в явном виде. То есть если число было выведено на экран или записано в файл в шестнадцатеричном формате, то все последующие числа также будут выводиться в шестнадцатеричном формате, пока не изменится флаг объекта `cout`.

Классы нового стиля

В ANSI C++ были внесены небольшие изменения для классов ввода/вывода. Тем не менее использование старых классов ввода/вывода, описываемых в этом разделе, остается возможным. Поэтому если их достаточно, то этот раздел можно пропустить.

- Чтобы переключиться с классов старого типа на классы ввода/вывода нового типа, надо выполнить два шага:

- заменить директиву `#include <iostream.h>` на `#include <iostream>`. Последняя версия не имеет расширения файла, так как обращается не прямо к файлу, а к его

виртуальному образу в стандартной библиотеке (см. также раздел «Новый стиль файлов заголовков» в приложении «Особенности ANSI C++»);

- ввести имена классов и объектов из пространства имен `std`:

```
using namespace std;
```

Второй шаг не обязателен и может быть опущен. Но тогда любая ссылка на класс ввода/вывода или на потоковый объект (такой как `cin` или `cout`) должна сопровождаться префиксом `std::`.

```
std::cout << "Введите, пожалуйста, число:";  
std::cin >> n;
```

После внесения указанных изменений можно использовать классы нового стиля практически так же, как старые классы. Другими словами, все примеры, описанные в этом разделе, будут работать и с новыми классами. Существуют, правда, и некоторые различия. Например, классы нового стиля поддерживают значительно больше флагов форматирования и манипуляторов ввода/вывода, различается также работа некоторых функций.

Обзор объектов и классов ввода/вывода C++

Далее в этом разделе кратко описываются классы и объекты ввода/вывода, такие как `cin` и `cout`. Помимо классов, описанных здесь, иерархия классов ввода/вывода также включает в себя некоторые абстрактные классы, такие как `ios` и `iostream`, объекты которых не могут быть созданы и их использование в явном виде невозможно (см. также раздел «Абстрактный класс» в главе «Словарь терминов и понятий»).

cerr

Объект `cerr` (консоль ошибок) предоставляет последовательный доступ к стандартному устройству вывода ошибок. Этот поток связан с монитором и обычно идентичен

стандартному потоку `cout`. Основное различие заключается в том, что даже если стандартный вывод направлен в файл, то все, что направляется в поток `cerr`, появляется на мониторе.

Объект `cerr` является объектом класса `ostream`. Список функций, которые могут быть вызваны через `cerr`, представлен в разделе «`ostream`».

```
cerr << "Ошибка с кодом ";  
cerr << error_number << endl;
```

В отличие от потока `clog`, поток `cerr` направляет информацию прямо на монитор, минуя какие-либо буферы. В остальном эти два потока одинаковы.

cin

Объект `cin` предоставляет последовательный доступ к стандартному устройству ввода информации. Этот поток обычно связан с клавиатурой. В системах типа UNIX, DOS или WINDOWS есть возможность перенаправить чтение информации с клавиатуры на чтение из файла.

Объект `cin` является объектом класса `istream`. Список функций, которые могут быть вызваны через `cin`, представлен в разделе «`istream`». Обычно `cin` используется как левый операнд оператора сдвига вправо (`>>`), работающего в качестве оператора ввода информации с объектами класса `istream`.

Например, введем три числа в переменные `a`, `b`, `c`:

```
int a, b, c;  
cin >> a >> b >> c;
```

Как и для всех объектов класса `istream`, числа, которые вводятся таким образом, должны быть разделены одним или несколькими пробелами, табуляциями или символами перехода на новую строку. Использование других разделителей, например запятой, может привести к ошибкам.

Функция `getline`, член класса `istream`, позволяет вводить строку, содержащую пробелы, что является альтернатив-

ным способом ввода нескольких чисел. Результат такого ввода затем можно проанализировать (см. также раздел «Функции работы со строками» в главе «Библиотечные функции»).

```
char input_string[72];  
cin.getline(input_string, 72);
```

clog

Объект `clog` предоставляет последовательный строковый доступ к монитору таким же образом, как объекты `cout` и `cerr`, и также принадлежит классу `ostream`.

Объекты `clog` и `cerr` практически одинаковы. Однако в отличие от потока `cerr`, поток `clog` использует буферизованный вывод. Оба потока (`cerr` и `clog`) являются предпочтительными устройствами для вывода сообщений об ошибках, так как они продолжают выводить сообщения на монитор даже в том случае, когда стандартный вывод (`cout`) перенаправлен (см. также разделы «`cerr`» и «`ostream`»).

cout

Объект `cout` предоставляет последовательный строковый доступ к устройству стандартного вывода. В системах типа UNIX, DOS или WINDOWS есть возможность перенаправить запись информации в файлы. В этом случае `cout` функционирует как файл.

Объект `cout` принадлежит классу `ostream`. Список функций, которые могут быть вызваны через `cout`, представлен в разделе «`istream`». Обычно `cout` используется как левый операнд оператора сдвига вправо (`<<`), работающего в качестве оператора вывода информации с объектами класса `ostream`.

Например, выведем значение числа `a` с предваряющим текстом:

```
cout << "Значение переменной a:";  
cout << a << endl;
```


Функции класса `ostream` используются для изменения формата вывода. Так, например, можно определить минимальную ширину поля вывода, в пределах которого следующее выводимое число будет выровнено по правой или левой границе.

fstream

Класс `fstream` поддерживает потоки с возможностью как ввода, так и вывода информации. Можно открыть поток для ввода или вывода, или ввода и вывода одновременно. Во избежание ошибок необходимо передавать содержимое буфера на физическое устройство перед переходом от операции чтения к операции записи (и наоборот) для одного и того же файла.

Для использования функций класса `fstream` в программу должна быть включена директива (при этом автоматически подключается и файл `iostream.h`):

```
#include <fstream.h>
```

Конструкторы класса `fstream` выглядят следующим образом:

```
fstream();  
fstream(char *filename, int mode, int  
    access=filebuf::openprot);  
fstream(filedesc fd);  
fstream(filedesc fd, char *pch, int nLength);
```

В первом случае конструктор создает пустой объект. Затем такой объект должен быть явно открыт путем вызова функции `open`.

Во втором случае конструктор создает объект `fstream` и открывает его, используя указанные параметры. Последний параметр `access` является необязательным и по умолчанию имеет значение `filebuf::openprot`, которое устанавливает нормальный режим доступа к файлам (список других режимов доступа можно узнать в документации компилятора).

В классах ввода/вывода нового стиля аргумент `access` отсутствует, а по умолчанию устанавливается текстовый режим ввода/вывода.

В следующем фрагменте объект `fio` класса `fstream` инициализируется как поток ввода/вывода в бинарном режиме:

```
fstream fio(fname, ios::in | ios::out | ios::binary);
```



ПРИМЕЧАНИЕ Напомним, что установки режима ввода/вывода являются флагами, и их можно комбинировать с помощью поразрядного оператора ИЛИ (`|`).

Третий и четвертый конструкторы используют дескриптор файла, который применяется на нижнем уровне, например, операционной системой. Четвертый конструктор определяет буфер, значение `NULL` для `pch` предотвращает буферизацию.

Для изучения функций-членов класса `fstream` см. также разделы «`ifstream`», «`ofstream`», «`istream`» и «`ostream`».

ifstream

Класс `ifstream` дает возможность открывать файлы как потоки, обращаться с файлом таким же образом, как с объектом `cin`.

Для использования класса `ifstream` в программу должна быть включена директива `#include <ifstream.h>`. При этом также автоматически включаются все описания из файла `iostream.h`.

Конструкторы `ifstream` выглядят следующим образом:

```
ifstream();  
ifstream(char *filename, int mode=ios::in, int  
    access=filebuf::openprot);  
ifstream(filedesc fd);  
ifstream(filedesc fd, char *pch, int nLength);
```

В первом случае конструктор создает пустой объект, который затем должен быть явно открыт путем вызова функции `open`.

Во втором случае конструктор создает объект `ifstream` и открывает его, используя указанные параметры. Последний параметр, `access`, является необязательным и по умолчанию имеет значение `filebuf::openprot`, которое устанавливает нормальный режим доступа к файлам (список других режимов доступа можно узнать в документации компилятора).

В классах ввода/вывода нового стиля аргумент `access` отсутствует, а по умолчанию устанавливается текстовый режим ввода/вывода.

Чтобы открыть файл для чтения в бинарном режиме, необходимо определить второй аргумент как `ios::binary`.

Третий и четвертый конструкторы используют дескриптор файла, который применяется на нижнем уровне, например, операционной системой. Четвертый конструктор определяет буфер, значение `NULL` для `rsh` предотвращает буферизацию.

В табл. 15 приведены основные флаги ввода/вывода.

Таблица 15. Флаги ввода/вывода

Флаг	Описание
<code>ios::app</code>	Открывает поток для записи информации в конец файла
<code>ios::ate</code>	Переходит к концу файла
<code>ios::binary</code>	Открывает файл в бинарном режиме (без преобразования символов новой строки)
<code>ios::in</code>	Открывает файл для ввода (чтения) из него
<code>ios::nocreate</code>	Открывает только уже существующие файлы
<code>ios::noreplace</code>	Работает только с новыми файлами. Если файл уже существует, операция не выполняется

Флаг	Описание
<code>ios::out</code>	Открывает файл для вывода (записи) в него
<code>ios::trunc</code>	Стирает все содержимое файла и открывает его как новый файл

В новых классах ввода/вывода (см. также раздел «Классы нового стиля») отсутствуют флаги `ios::noreplace` и `ios::nocreate`. Все остальные сохранены.

В том случае, когда конструктору не удалось открыть поток (файл), результатом будет объект `ifstream` нулевой длины. Конструктор откроет этот поток (считая его длину нулевой), даже если файл не существует, если не использован флаг `ios::nocreate` (открытие файла только в том случае, если он существует).

Следует проверять, был ли открыт файл, прежде чем выполнять с ним какие-либо операции, например, так:

```
ifstream myfin("c:\\work\\param.txt", ios::in |  
    ios::nocreate);  
if (!myfin)  
    cout << "Ошибка открытия файла параметров"
```

Класс `ifstream` наследует все функции класса `istream`. Помимо них он также содержит функции, описанные в табл. 16 (функции `seekg`, `tellg` и `read` принадлежат классу `ostream`, но кроме как с файлами и буферами строк они практически нигде не используются).

Таблица 16. Функции класса `ifstream`

Функция	Описание
<code>close()</code>	Закрывает поток
<code>gcount()</code>	Возвращает количество введенных последний раз символов
<code>open(fname, mode)</code>	Открывает поток, используя те же параметры, что и конструктор <code>ifstream</code>

продолжение ➤

Таблица 16. Функции класса `ifstream`

(продолжение)

Функция	Описание
<code>read(buf, num)</code>	Считывает <code>num</code> символов (или все символы до конца файла, если их меньше <code>num</code>) и возвращает ссылку на текущий поток ввода, <code>buf</code> является массивом символов
<code>seekg(offset, org)</code> <code>seekg(position)</code>	Перемещает указатель ввода. В первом случае на <code>offset</code> байт, считая от одной из трех позиций, определяемых параметром <code>org</code> : <code>ios::beg</code> , <code>ios::cur</code> или <code>ios::end</code> . Вторая функция переводит указатель в положение <code>position</code> . Параметры <code>offset</code> и <code>position</code> имеют тип <code>streamoff</code> и <code>streampos</code> соответственно
<code>tellg()</code>	Возвращает позицию указателя ввода файла, имеющую тип <code>streampos</code> , определенный в файле <code>iostream.h</code>

Операции чтения из файлов выполняются путем открытия объекта класса `ifstream` и дальнейшего чтения из него, подобно чтению из объекта `cin`.

Программа, приведенная ниже, позволяет вывести на экран содержимое файла. Каждая строка файла на экране выводится после символа `>`.

```
#include <iostream.h>
#include <fstream.h>
int main(void)
{
    char text[81];
    cout << "Введите имя файла:";
    cin >> text;
    ifstream fin(text, ios::in | ios::nocreate);
    if (!fin)
```



```
{
    cout << "Файл открыть не удалось";
    return 1;
}
while (!fin.eof())
{
    fin.getline(text, 81);
//в тот же массив читаем строки
    cout << ">" << text << endl;
}
getch();
return 0;
}
```

Отличие между текстовым и бинарным режимами в том, что в бинарных файлах нет символов перехода на новую строку. В остальном эти режимы одинаковы, и для ввода информации из файла в бинарном режиме можно использовать приведенный выше пример с одним изменением:

```
ifstream fin(text, ios::in | ios::binary);
```

Функция `read` читает байты прямо в символьный массив, объявленный как `char`, `unsigned char` или `signed char`. Можно определить количество введенных символов с помощью функции `gcount`. Если количество введенных символов меньше требуемого, значит, обнаружен конец файла.

istream

Класс `istream` является базовым для классов `ifstream` и `istrstream`, поэтому все его члены являются членами этих классов. В свою очередь, он является классом, производным от абстрактного базового класса `iostream`, и наследует все его члены.

В табл. 17 перечислены основные функции-члены класса `istream`. Большинство из них имеет дело со статусом и флагами формата. Следует напомнить, что при определении параметров знак равенства (=) обозначает значение по умолчанию. Этот параметр может быть опущен во время вызова функции.

Таблица 17. Основные функции ввода/вывода

Функция	Описание
<code>bad()</code>	Возвращает ненулевое значение, если возникла фатальная ошибка
<code>clear(flags = 0)</code>	Очищает флаги форматирования, если параметры не определены, иначе устанавливает указанные флаги в соответствии с параметром <code>flags</code>
<code>eof()</code>	Возвращает ненулевое значение, если достигнут конец файла
<code>fail()</code>	Возвращает ненулевое значение, если возникла ошибка
<code>flags flags(f)</code>	Первый вариант функции возвращает текущие значения флагов. Второй вариант функции использует параметр <code>f</code> (типа <code>long</code>) в качестве новых значений флагов и возвращает предыдущие настройки, также типа <code>long</code>
<code>good()</code>	Возвращает ненулевое значение при отсутствии ошибок
<code>rdstate()</code>	Возвращает статус последней операции ввода/вывода (возможные значения представлены ниже)
<code>setf(flags)</code> <code>setf(flags1, flags2)</code>	Первая функция устанавливает все флаги форматирования, которые указаны в параметре <code>flags</code> . Вторая функция устанавливает те биты в <code>flags1</code> , которые указаны в <code>flags2</code>
<code>sync_with_stdio()</code>	Дает возможность использовать классы ввода/вывода вместе с функциями из файла <code>stdio.h</code> (<code>fopen</code> , <code>scanf</code> и другими)

Функция `rdstate` сообщает состояние последней операции ввода/вывода потока. Возможны следующие значения:

- `ios::goodbit` — отсутствие ошибок;
- `ios::eofbit` — был достигнут конец файла (EOF);

- ✱ `ios::failbit` — возникла не фатальная ошибка;
- ✱ `ios::badbit` — возникла фатальная ошибка.

Рассмотрим пример:

```
cin.read (buf, 10);  
if (cin.rdstate() == ios::eofbuf)  
//. . . конец чтения;
```

В табл. 18 перечислены функции класса `istream` для ввода информации (некоторые из них, такие как `seekg`, `gcount` и `read`, работают только с файлами бинарного типа и определены в классе `ifstream`). Напомним, что при определении параметров знак равенства (=) обозначает значение по умолчанию. Такой параметр может быть опущен во время вызова функции.

Таблица 18. Функции ввода класса `istream`

Функция	Описание
<code>eatwhite()</code>	Пропускает начальные пробелы
<code>get(ch)</code> <code>get(buf, num, delim='\n')</code> <code>get(buf, delim='\n')</code>	Первый вариант получает ссылку на символ <code>ch</code> и считывает символы, начиная со следующего после него. Второй вариант считывает до <code>num</code> символов и копирует их в строку <code>buf</code> типа <code>char*</code> . Третий вариант получает ссылку на другой потоковый объект <code>buf</code> . Все три версии функции возвращают ссылку на текущий поток ввода
<code>getline(buf, num, delim='\n')</code>	Считывает до <code>num-1</code> символов в буфер <code>buf</code> типа <code>char *</code> , завершая строку нуль-терминатором. Возвращает ссылку на текущий поток ввода

продолжение ➤

Таблица 18. Функции ввода класса `istream` (продолжение)

Функция	Описание
<code>ignore(num=1, delim=EOF)</code>	Читает и игнорирует символы до тех пор, пока не будет прочитано <code>num</code> символов или не встретится разделитель, определенный параметром <code>delim</code> . Возвращает ссылку на текущий поток ввода
<code>peek()</code>	Возвращает следующий символ без удаления его из потока. Выводит символ <code>EOF</code> , если достигнут конец файла

Для выполнения почти всех операций ввода, поддерживаемых функциями из файла `stdio.h`, можно использовать функции-члены класса `istream` и производных от него классов `ifstream` и `istrstream`.

Рассмотрим пример:

```
int num = 0;
char wstr[81];
cout << "Введите десятичное число:";
cin >> num;
cout.setf (ios::hex);
cout << "соответствует 16-ричному" << num << endl;
cin.getline (wstr, 81);
//ввод символа новой строки
//можно использовать cin.ignore();
cout << "Для продолжения нажмите ENTER.";
cin.getline (wstr, 81);
cin.getline(text, 81); //чтение до конца строки
cin >> text; //чтение до очередного пробела
```



ПРИМЕЧАНИЕ Напомним, что есть очень важное различие между вызовом функции `getline` и использованием оператора сдвига вправо (`>>`). В первом случае читаются все символы до перевода строки (нажатия клавиши Enter), во втором — до первого пробела.

istream

Класс `istream` позволяет читать данные из строк в памяти так же, как с консоли или из файла, и является аналогом функции `sscanf` из файла `stdio.h`.

Конструкторы класса `istream` выглядят следующим образом:

```
#include <istrstream.h>
istream();
istream(char *pch, int nLength);
```

В первом случае конструктор создает поток, используя внутренний буфер строк, который сначала пуст, но расширяется и сокращается в процессе выполнения программы. Вторым конструктор, который является более предпочтительным для большинства операций, использует в качестве буфера строку определенной длины.

Класс `istream` поддерживает все функции класса `istream`, включая `read`, `seekg` и `tellg`. Также класс поддерживает функцию `str` (см. также раздел «`istrstream`»).

ofstream

Класс `ofstream` предоставляет возможность открывать файлы для записи подобно потокам вывода. С помощью `ofstream` можно записывать символьную информацию в файлы таким же образом, как на экран с помощью объекта `cout`. Также выполняется запись в файлы в бинарном режиме.

Конструкторы класса `ofstream` выглядят следующим образом:

```
#include <fstream.h>
ofstream();
ofstream(char *filename, int mode=ios::out, int
    access=filebuf::openprot);
ofstream(filedesc fd);
ofstream(filedesc fd, char *pch, int nLength);
```

В первом случае конструктор создает пустой объект, который затем должен быть явно открыт путем вызова функции `open`.

Второй вариант является более применяемым. Конструктор создает объект `ofstream` и открывает его, используя указанные параметры. Последний параметр, `access`, является необязательным и по умолчанию имеет значение `filebuf::openprot`, которое указывает на нормальный режим доступа к файлам (список других режимов доступа можно узнать в документации компилятора).

В классах ввода/вывода нового стиля параметр `access` отсутствует, и по умолчанию устанавливается текстовый режим ввода/вывода. Чтобы открыть файл для чтения в бинарном режиме, необходимо определить второй аргумент как `ios::binary`. Этот флаг можно комбинировать с флагом `ios::out` с помощью оператора ИЛИ (`|`). Если конструктору не удастся открыть определенный файл, то результатом будет нулевой объект класса `ofstream`.

Третий и четвертый конструкторы используют дескриптор файла, который применяется на нижнем уровне, например, операционной системой. Четвертый конструктор определяет буфер, значение `NULL` для `rch` предотвращает буферизацию.

Таблица 19. Функции класса `ofstream`

Функция	Описание
<code>close()</code>	Закрывает поток
<code>open(fn, mode)</code>	Открывает поток, используя параметры конструктора <code>ofstream</code>
<code>seekp(offset, org)</code> <code>seekp(position)</code>	Первый вариант сдвигает указатель на смещение <code>offset</code> относительно одного из трех значений: <code>ios::beg</code> , <code>ios::cur</code> , <code>ios::end</code> , указанных параметром <code>org</code> . Второй вариант перемещает указатель в позицию <code>position</code> . Параметры <code>offset</code> и <code>position</code> имеют тип <code>streamoff</code> и <code>streampos</code> соответственно

Функция	Описание
tellp()	Возвращает позицию указателя, имеющую тип streampos
write(buf, num)	Записывает num байт в массив символов buf

Класс `ofstream` наследует все функции-члены класса `ostream`. Помимо них он поддерживает функции, перечисленные в табл. 19. Заметим, что функции `seekp`, `tellp` и `write`, объявленные в классе `ostream`, в действительности используются только с файлами и буферами строк.

Как только объект класса `ofstream` открыт, его можно использовать как текстовый файл, аналогично объекту `cout`. В следующем примере вещественные числа считываются из одного файла и записываются, умноженные на 5, в другой файл:

```
#include <fstream.h>
int main(void)
{
    char infn[81], outf[81];
    cout << "Введите имя файла для ввода:";
    cin >> infn;
    cout << "Введите имя файла для вывода:";
    cin >> outf;
    ifstream fin(infn, ios::in | ios::nocreate);
    ifstream fout(outf);
    if (!fin || !fout)
    {
        cout << "Не открыть один из файлов";
        return 1;
    }
    float fw;
    while (!fin.eof())
    {
        fin >> fw; //чтение fw из файла ввода
        fout << fw*5; //запись fw*5 в файл вывода
    }
    return 0;
}
```




ПРИМЕЧАНИЕ Напомним, что операторы сдвига (<< и >>) записывают и считывают информацию только в текстовом формате. И в примере, приведенном выше, вещественные числа считываются как строки и затем преобразуются в вещественные значения. При выводе, наоборот, вещественные значения, умноженные на 5, преобразуются в строки, и эти строки записываются в файл. Таким образом, все действия аналогичны действиям при вводе с клавиатуры и выводе на экран.

Для работы с файлами в бинарном режиме нужно использовать следующие инструкции:

```
ofstream fout(fn, ios::out | ios::binary);  
float fw = 173.258;  
fout.write((char*)&n, sizeof(n)); /*вывод в файл n в виде  
числа*/
```

Информация, которая записана с помощью функции `write`, может быть прочитана из потока ввода с помощью функции `read`.



СОВЕТ При работе с большими массивами чисел рекомендуется использовать бинарный режим записи в файл, так как он более компактен. Например, число 173.258 (если оно объявлено как `float`) занимает 4 байта в бинарном виде и 8 байт — в текстовом.

ostream

Класс `ostream` определяет функции, которые вызываются с объектами `cout`, `cerr`, `clog`. Также `ostream` является базовым классом для классов `ofstream` и `ostrstream`.

Класс `ostream` унаследовал все общие функции обработки и флаги форматирования (табл. 20).

Функции `precision` и `width` устанавливают свойства формата печатного поля для следующей операции. Не обязательно устанавливать эти параметры. Фактически, по умолчанию печатное поле принимает ширину, необходимую для вывода переменной.

Таблица 20. Функции вывода и форматирования класса ostream

Функция	Описание
fill() fill(ch)	В первом случае возвращается текущий символ заполнения (по умолчанию — пробел). Во втором случае устанавливается символ заполнения ch, и возвращается его предыдущее значение. Чтобы записывать нули в начале числа, следует определить 0 как символ заполнения
flush()	Записывает содержимое потока вывода на физическое устройство
precision() precision(p)	В первом случае возвращается текущая точность. Вторая версия устанавливает новую точность и возвращает старую. Эта величина определяет количество выводимых цифр. Последняя цифра числа с плавающей точкой округляется
put(ch)	Выводит в поток символ ch и возвращает ссылку на поток вывода
putback(ch)	Передаёт символ ch обратно в поток вывода и возвращает ссылку на него. Символ ch должен быть записан последним, иначе возникнет ошибка
width() width(w)	В первом случае возвращается текущая ширина поля вывода. Вывод, который меньше чем установлено, дополняется символом, определенным с помощью функции fill(ch). Вторая версия устанавливает новую ширину и возвращает старую. Эта величина определяет ширину поля для следующей операции вывода

Например, можно выводить числа таким образом:

```
cout.width(5);  
cout << 5.5 << endl;  
cout.width(5);  
cout << 55.5555;
```


При этом на экране будет:

5.5

55.556



ПРИМЕЧАНИЕ Второе число не входит в ширину поля, потому что имеет точность пять цифр. Указав точность, можно округлить его до пяти знаков.

```
cout.precision(4)
cout.width(5);
cout << 55.5555;
```

В таком случае мы увидим:

55.56

ostream

Класс `ostream` позволяет писать строки в память, как в консоль или файл, и является аналогом функции `sprintf`, определенной в файле `stdio.h`.

Конструкторы класса `ostream` выглядят следующим образом:

```
#include <ostream.h>
ostream();
ostream(char *pch, int nLength, int mode=ios::out);
```

В первом случае конструктор создает поток, используя внутренний буфер строк, который сначала пуст, но расширяется и сокращается в процессе выполнения программы. Второй конструктор, который является более предпочтительным для большинства операций, использует в качестве буфера строку определенной длины.

Класс `ostream` поддерживает все функции класса `ostream` (см. также раздел «`ostream`») и функции `pcount` и `str` класса `ostream` (см. также раздел «`ostream`»).

istream

Класс `istream` и его производные классы `istream` и `ostream` позволяют читать и писать строки в память так

же, как в файлы или на консоль. Такая возможность аналогична возможностям функций `sprintf` и `sscanf`, определенным в файле `stdio.h`.

Конструкторы класса `strstream` выглядят следующим образом:

```
#include <strstrea.h>
strstream();
strstream(char* pch, int n, int mode);
```

В первом случае конструктор создает поток, используя внутренний буфер строк, который сначала пуст, но расширяется и сокращается в процессе выполнения программы. Второй конструктор, который является более предпочтительным для большинства операций, использует в качестве буфера строку определенной длины.

Класс `strstream` согласуется со всеми функциями классов `istream` и `ostream`, включая функции `read`, `write`, `gcount`, `seekg`, `seekp`, `tellg` и `tellp` (см. также разделы «`istream`» и «`ostream`»). Помимо них класс `ostrstream` поддерживает функции, перечисленные в табл. 21 (знак равенства (=) обозначает параметр со значением по умолчанию).

Таблица 21. Функции класса `ostrstream`

Функция	Описание
<code>freeze(f=true)</code>	Если значение <code>f</code> равно <code>true</code> , функция замораживает размер и положение строки данных. Если же значение <code>false</code> , функция размораживает информацию. Эта операция работает только с динамическим буфером строк
<code>pcount()</code>	Возвращает количество символов, которые хранятся в строке
<code>rdbuf()</code>	Возвращает указатель на объект <code>strstreambuf</code> , который может быть использован для выполнения операций низкого уровня в буфере строки

продолжение ↗

Таблица 21. Функции класса `ostrstrea` (продолжение)

Функция	Описание
<code>str()</code>	Возвращает указатель на строку типа <code>char*</code> . С динамическими строками функция имеет эффект замораживания данных строки

Если поток использует внутренний, динамический буфер, выделенный с помощью первого варианта конструктора, то эффект вызова функции `str` будет аналогичен замораживанию строки. То есть ее размер и положение будут фиксированными для последующих операций. Можно разморозить строку путем вызова функции `freeze` с параметром `false`, как в следующем примере:

```
#include <ostrstrea.h>
#include <string.h>
//. . .
ostrstream str_io;
str_io << "Пример";
char *p = str_io.str(); //заморозили
//. . .
char strdata[256];
strncpy (strdata, 255, p); //сохранили
//. . .
str_io.freeze (false); //разморозили
//после этого значение p не определено
//. . .
p = str_io.str(); //получили новое значение
```

Если введенный текст в последующем размораживается, то лучше сохранить данные, потому что после того, как данные размораживаются, указатель `p` принимает неизвестное значение, пока не будет повторно вызвана функция `str`.

Часть II

Краткий курс программирования на C++

Основные особенности программирования на C++

Эта глава будет особенно полезна начинающим программистам. В ней описываются первые шаги программирования на C++: структура программы, необходимые инструкции и директивы, некоторые полезные советы и детали, на которые следует обратить особое внимание. В главе вводятся понятия переменной, функции и области их действия (видимости).

Нужно отметить, что C и C++ — два разных языка программирования, но они имеют много общих базовых понятий. Для простоты в этой главе мы будем говорить о языке C++, отмечая его отличия от C там, где они есть.

Первая программа на C++

В качестве задачи для первой программы возьмем вывод обычной строки на экран монитора.

Пример 1

```
#include <stdio.h>
void main ()
{
    printf ("\nЗдравствуй, мир!");
}
```

В результате работы программы на экране монитора должна появиться строка:

Здравствуй, мир!

Обсудим действия, выполняемые каждой из строк программы:

- `#include <stdio.h>` — директива, необходимая для того, чтобы функция `printf` (вывод строки на экран монито-

ра) была известна программе (см. также главу «Директивы, макросы и операторы препроцессора»);

- `void main ()` — заголовок программы;
- `{` и `}` — признаки начала и конца блока (группы) инструкций. Программа — это тоже блок инструкций;
- `printf ("\nЗдравствуй, мир!");` — единственная исполняемая инструкция этой программы, выводящая строку на экран монитора;
- `\n` — управляющий символ, который выполняет перевод курсора на экране в начало следующей строки. Такие управляющие выводом последовательности символов, начинающиеся с символа обратный слэш (`\`), называются управляющими последовательностями или Esc-последовательностями (см. главу «Типы данных»);
- точка с запятой (`;`) — признак конца инструкции, его наличие обязательно для каждой инструкции.

Язык C++ является позиционно-независимым. Это значит, что инструкции и операнды могут располагаться в любой позиции строки и между ними может быть любое количество пробелов и табуляций. Отсутствие пробелов также не является ошибкой.

Для простоты чтения программы рекомендуется набирать каждую инструкцию блока в новой строке, но с одной и той же позиции, сдвигая их относительно признака начала блока (символа `}`), при этом символ `}` следует располагать в той же позиции, что и соответствующий ему символ `{`. В этом случае блок инструкций будет выглядеть так:

```
{  
    инструкция1;  
    инструкция2;  
    инструкция3;  
}
```



ВНИМАНИЕ В C++ прописные и строчные буквы различаются, то есть `a` и `A` — разные символы. Для названий библиотечных функций, как правило, используются строчные буквы, а для обозначения констант — прописные.



ПРИМЕЧАНИЕ Создание программы можно разделить на три основных этапа: набор текстового файла, его компиляцию и построение исполняемого файла (программы). Детали этого процесса зависят от используемой среды программирования.

Объявление переменных

Практически все программы должны тем или иным образом обрабатывать информацию. Но прежде чем заняться обработкой данных, необходимо выделить место для их размещения в оперативной памяти компьютера. Этот процесс называется объявлением переменных.

Размер выделяемой памяти зависит от типа переменной: для целых чисел — два байта, для длинных целых и чисел с плавающей точкой — четыре байта и т. д. (см. также главу «Типы данных»).

В C++ инструкция объявления переменных состоит из строки, начинающейся с имени типа (`int`, `float`, `long`), за которым следует имя переменной. Как обычно, в конце инструкции ставится признак ее окончания — точка с запятой (;).

В одной строке может быть объявлено несколько переменных одного типа, в этом случае они разделяются запятыми:

```
int i;  
char wchar;  
short p, j, iw;
```

При объявлении можно одновременно инициализировать переменные (присваивать им начальные значения). Для этого после переменной ставится знак равенства и значение, присваиваемое переменной:

```
int x0 = 10, step = 1;
```

Пример 2

```
#include <stdio.h>  
void main ()
```



```
{
    int r = 1;
    float pi = 3.14;
    printf ("\nДлина окружности с радиусом %dсм равна
%7.2fсм.", r, 2 * pi * r);
}
```

В результате работы программы на экране монитора должна появиться строка:

Длина окружности с радиусом 1см равна 6.28см.

Функция вывода данных на экран монитора printf поддерживает форматированный вывод. Спецификатор %d означает вывод следующего аргумента в виде десятичного целого, спецификатор %7.2f — вывод следующего аргумента в виде числа с плавающей точкой, занимающего 7 позиций, включая десятичную точку, с двумя знаками после запятой (см. также главу «Объекты и классы библиотеки ввода/вывода»).

Все простейшие типы данных являются различными вариациями двух типов: целого и с плавающей точкой. Числа с плавающей точкой могут содержать дробную часть, целые не могут. Использование чисел с плавающей точкой является более гибким, целых чисел — более эффективным. Если точно известно, что данные не могут содержать дробную часть, лучше объявить их переменными целого типа.

Основные характеристики типов short, long, int, float, double приведены в табл. 22 (см. также главу «Типы данных»).

Таблица 22. Характеристики типов данных C++

Тип	Название	Размер	Формат функции printf	Диапазон
Int	Целое	2 байта	%d	От -32 768 до 32 767
Unsigned	Целое	2 байта	%ud	от 0 до 65 535

продолжение ➤

Таблица 22. Характеристики типов данных C++ (продолжение)

Тип	Название	Размер	Формат функции printf	Диапазон
Short	Целое	2 байта	%d	От -32 768 до 32 767
Long	Целое	4 байта	%ld	От -2 147 483 648 до 2 147 483 647
Float	Число с плавающей точкой	4 байта	%f	От $-3.4 \cdot 10^{+38}$ до $-3.4 \cdot 10^{-38}$ и от $3.4 \cdot 10^{-38}$ до $3.4 \cdot 10^{+38}$
Double	Число с плавающей точкой	8 байт	%lf	От $-1.7 \cdot 10^{+308}$ до $-1.7 \cdot 10^{-308}$ и от $1.7 \cdot 10^{-308}$ до $1.7 \cdot 10^{+308}$

Директива #include

В C++ не только переменные, но и все функции, за исключением функции main, должны быть объявлены до их использования. Объявление функции является способом сообщить компьютеру, параметры каких типов функция будет получать и какое значение возвращать. Явное объявление требуется не только для новых функций, написанных программистом, но и для функций из стандартных библиотек. Для этой цели создаются файлы заголовков (header files), которые содержат объявления (заголовки) всех нужных программисту функций. Заголовки всех стандартных функций ввода/вывода содержатся в файле stdio.h. Директива #include — директива чтения файла заголовков, которая позволяет объявить все стандартные функции.

Директива — специальная команда, которая выполняется до того, как начнется компиляция программы. Синтаксис директивы отличен от синтаксиса инструкции: она всегда начинается со знака # в первой позиции строки и не требует никакого признака окончания.

В табл. 23 приводятся некоторые файлы, содержащие заголовки стандартных функций C++. Любой из этих файлов может быть включен в программу с помощью директивы `#include <имя файла>`.

Таблица 23. Стандартные файлы заголовков C++

Имя файла	Описание функций
Stdio.h	Стандартные функции ввода и вывода, включая функции работы с файлами
iostream.h	Операторы работы с потоками (только для C++), которыми могут быть заменены функции <code>printf</code> и <code>scanf</code> (см. также раздел «Операторы потоков для работы с файлами» в главе «Ввод и вывод»)
string.h	Функции работы со строками
math.h	Тригонометрические, логарифмические и другие математические функции
malloc.h	Функции динамического распределения и освобождения памяти (в C++ также есть встроенные операторы <code>new</code> и <code>delete</code> для этой цели)

Ввод данных

Действия, которые обычно выполняются с данными, можно разделить на три основные группы: ввод данных, присвоение им новых значений после некоторых вычислений и вывод данных.

Для ввода данных с клавиатуры используется функция `scanf`. Как и функция `printf`, она поддерживает форматированный вывод. Однако важной особенностью функций ввода C++ является косвенная адресация памяти, поэтому к переменной, значение которой вводится, должен быть применен оператор определения адреса амперсанд (&).

Подсказка для ввода данных может быть выполнена с помощью функции `printf`.

Пример 3

```
#include <stdio.h>
void main()
{
    int r;
    float pi = 3.14;
    printf ("\nВведите значение радиуса:");
    scanf ("%d", &r);
    printf ("\nДлина окружности с радиусом %dсм равна
    %7.2fсм.", r, 2 * pi * r);
}
```

После запуска программы на экране монитора должна появиться строка:

Введите значение радиуса:

Введите число, например 10, и нажмите клавишу Enter (нажатие клавиши Enter является признаком окончания ввода для функции `scanf`). После этого на экране монитора должна появиться строка:

Длина окружности с радиусом 10см равна 62.80см.



ВНИМАНИЕ Наиболее распространенной ошибкой при работе с функциями ввода C++ является пропуск оператора определения адреса (&).

Операции, выражения и комментарии

Операции в C++ делятся на три основные группы: арифметические, логические и операции сравнения.

Наиболее простыми арифметическими операциями в C++ являются присваивание (=), сложение (+), вычитание (-), умножение (*), деление (/), определение остатка от деления (%).

К операциям сравнения относятся > (больше), < (меньше), == (равно), >= (больше или равно), <= (меньше или равно), != (не равно).

К простым выражениям относятся константа, переменная и вызов функции. Составные выражения образованы из простых выражений, которые соединены между собой операторами (знаками операций). Порядок вычисления выражений определяется правилами приоритета. Изменить приоритет выражения можно с помощью скобок (см. раздел «Приоритет» в главе «Словарь терминов и понятий»).

Кроме того, по числу операндов (выражений, участвующих в операции) операции делятся на унарные и бинарные. К унарным операциям относятся изменение и явное указание знака (унарный минус и унарный плюс), логическое и поразрядное отрицание (см. также раздел «Логические операции и операции над битами»), инкремент и декремент (см. также раздел «Особые операции C++») и некоторые другие. Большая часть операций относится к бинарным, в частности, это все приведенные в начале этого раздела арифметические операции и операции сравнения.

Рассмотрим структуру некоторых инструкций. В левой части инструкции присваивания может находиться только переменная. Далее следует знак равенства (=) и выражение.

```
a = 67.3; b = 37;  
a1 = a * b + 125.3;  
a2 = a * ((b + 125.3) / 183.66);
```

Отличительной особенностью C++ является то, что инструкция присваивания сама является выражением. Это означает, что ее, как и любое другое выражение, можно использовать в правой части инструкции присваивания.

```
x1 = x2 = y1 = y2 = a+b;
```

Присваивание при этом выполняется справа налево следующим образом:

- y_2 присваивается значение $a+b$;
- вычисляется значение выражения $y_2=a+b$, это $a+b$;
- y_1 присваивается значение вычисленного выражения, то есть $a+b$;
- вычисляется значение выражения $y_1=a+b$, это $a+b$;

- $x2$ присваивается значение вычисленного выражения, то есть $a+b$;

и так далее.

Инструкция сравнения также является выражением и состоит из двух выражений, между которыми стоит знак операции (оператор) сравнения.



ВНИМАНИЕ Перед выполнением каких-либо операций над переменными они должны быть объявлены.



СОВЕТ Для удобства чтения программы, особенно при наличии большого количества операторов в одной инструкции, может оказаться полезным отделение друг от друга пробелами операндов и операторов.

Для пояснения текста программы можно использовать комментарии. Они не являются выполняемыми инструкциями и могут выделяться двумя способами: вставкой двойного слэша перед комментарием (//) или выделением комментария символами (/*) и (*/*). В первом случае концом комментария считается конец строки.

```
c = 2*pi*r; //вычисление длины окружности
/* pi должно быть инициализировано раньше*/
```

Функции

Повторяющуюся группу инструкций программы выделяют в отдельную подпрограмму — функцию. В программах, текст которых занимает больше одного экрана монитора, также рекомендуется выделить отдельные функции.

Функция — это логически самостоятельная часть программы со своим именем, которой могут передаваться некоторые переменные значения (параметры), и которая может возвращать какое-то значение. Тип этого значения и типы параметров указываются в заголовке функции. Если в качестве типа функции указан тип `void`, функция не возвращает никакого значения.

Прежде чем использовать функцию, она должна быть объявлена. Инструкция объявления, которая называется прототипом, состоит из строки, начинающейся с типа функции (тип значения, которое она возвращает), за которым следует имя функции, список параметров, заключенный в скобки, и признак окончания инструкции (;). Список параметров состоит из объявлений каждой переменной, отделенных друг от друга запятыми.

Примеры прототипов функций:

```
// Функция вычисления суммы двух целых:  
int sum (int a, int b);  
// Функция вывода сообщения на экран:  
void myprint (void);
```

При вызове функции вместо формальных параметров, указанных в ее прототипе, подставляются фактические значения: переменные или константы.

И наконец, сами инструкции функции (ее тело) оформляются в виде блока, то есть заключаются в фигурные скобки {}. Перед блоком инструкций записывается заголовок определения функции, который отличается от прототипа отсутствием признака окончания инструкции (;). Тело функции, заключенное в фигурные скобки {}, называют определением функции.

 **ВНИМАНИЕ** После определения функции точка с запятой не ставится.

Выход из функции выполняется инструкцией `return;`. Ее можно опустить, если тип функции — `void`. В остальных случаях эта инструкция формирует результат работы функции и имеет вид: `return (выражение);`. Тип выражения и тип функции должны совпадать.

Преобразуем текст последней программы с помощью функции. Результат работы программы не изменится. Однако преимуществом такого подхода является возможность использования этой же функции в другой программе без каких-либо изменений.

Пример 4

```
#include <stdio.h>
float myc (float r);
/*Прототип функции вычисления длины окружности*/
void main()
{
    float r, res;
    printf ("\nВведите значение радиуса:");
    scanf ("%f", &r);
    res = myc (r); //Вызов функции
    printf ("\nДлина окружности с радиусом %5.0fсм равна
    %7.2fсм.", r, res);
}
float myc (float r)
/*Определение функции вычисления длины окружности*/
{
    float pi = 3.14;
    return (2 * pi * r);
}
```



СОВЕТ Прототип функции отличается от заголовка ее определения только наличием знака окончания инструкции (;), поэтому рекомендуется просто копировать заголовок в прототип или наоборот, соответственно добавляя или удаляя точку с запятой (;). Так удастся избежать довольно распространенной ошибки — несоответствия прототипа и определения функции.



ПРИМЕЧАНИЕ В C++ функция `main` отличается от всех остальных функций только тем, что она является точкой входа в программу, то есть никакая другая функция ее не вызывает, и, соответственно, она не нуждается в прототипе.

Локальные, глобальные и другие переменные

Одним из атрибутов переменной является область ее действия, или, иначе, область видимости. В C++ существуют

четыре вида (класса) переменных: локальные, глобальные, статические и внешние. Все они унаследованы из языка С.

Локальная переменная объявляется внутри определения функции. В каждой функции можно определить переменную с одинаковым именем, например `i`. Эта переменная может изменяться в теле функции, не вызывая изменений переменных с тем же именем в других функциях. При выходе из тела функции ее значение не определено. Этим же свойством обладают локальные переменные, объявленные внутри функции `main`.

Предпочтительнее объявлять переменную локальной (в этом случае значительно легче определить, в какой инструкции она изменяется), но иногда необходимо расширить область действия переменной за пределы одной функции. В таком случае используются *глобальные переменные*, распространяющие область своего действия на весь исходный файл, начиная с места объявления. Глобальные переменные реализуют механизм разделенного доступа к информации и сохраняют свои значения в течение всего времени работы программы.

Чтобы сделать переменную глобальной, надо объявить ее вне всех определений функций. Чаще всего это делается в начале файла, например, сразу после директив.

Статическая переменная (`static`) сочетает область видимости локальной переменной с расширенным временем жизни глобальной переменной. Это может быть полезно, если нужно, чтобы локальная переменная сохраняла свое значение между вызовами функции, но не могла быть изменена другими функциями. Статическая переменная определяется в теле функции обычной инструкцией объявления (определения) переменной со словом `static` перед ним:

```
static int snum;
```

Очень часто большие программы стараются разделить на несколько модулей (файлов). Каждый модуль компилируется отдельно, а потом с помощью проектного файла и компоновщика все они собираются в одну программу. Каждая функция автоматически видна всем другим модулям

проекта до тех пор, пока она не объявлена статической (это делается размещением ключевого слова `static` перед ее прототипом и заголовком в определении функции). Статическая функция видна только в пределах модуля, начиная с места объявления.

Переменные, наоборот, видны только внутри модуля до тех пор, пока они не объявлены как внешние (`external`). Чтобы сделать переменную внешней, надо объявить ее глобальной в одном, и только одном, модуле. Во все остальные модули, обращающиеся к этой переменной, следует включить то же самое объявление, но с предшествующим словом `extern`:

```
extern int gnum;
```

Другим способом использования таких переменных является включение всех внешних объявлений в один файл заголовков, а затем включение полученного файла заголовков во все исходные модули. Но и в этой ситуации каждая из переменных должна быть объявлена как глобальная в одном, и только одном, модуле.

Управляющие инструкции

В C++ к *управляющим инструкциям* относятся инструкции `if`, `while`, `do`, `for` и `switch`. В этом разделе описываются две из них: условная (`if`) и одна из инструкций цикла (`while`).

Синтаксис условной инструкции:

```
if (выражение) инструкция1  
else инструкция2
```

Если выражение, заключенное в круглые скобки, принимает значение, не равное нулю (истина), выполняется инструкция1. Если оно равно нулю (ложь), выполняется инструкция2.

Вторая строка инструкции `if` является не обязательной и может отсутствовать, тогда условная инструкция имеет вид:

```
if (выражение) инструкция1
```


Следует отметить, что инструкция1 и инструкция2 могут быть *составными*, то есть представлять из себя блок инструкций (группу инструкций, заключенных в фигурные скобки {}). Если эти инструкции являются простыми, они должны заканчиваться знаком окончания инструкции (;):

```
if (i <= 10) x = 0;
if (a == b)
{
    c1 = c1 + 1;
    z = 77;
}
else c2 = c2 + 1;
```



ПРИМЕЧАНИЕ В C++ значению ИСТИНА соответствует не только значение 1, но и любое другое ненулевое значение. Инструкция сравнения принимает значение 0 (ЛОЖЬ) или 1 (ИСТИНА).



ВНИМАНИЕ В выражении условной инструкции значительно чаще требуется операция проверки равенства, которая отличается от операции присваивания. Первая реализуется двойным знаком равенства, то есть оператором (==), а вторая — одинарным, то есть оператором (=). Использование оператора (=) вместо оператора (==) является одной из наиболее распространенных ошибок.

Синтаксис инструкции цикла while:

```
while (выражение) инструкция1
```

До тех пор пока выражение, заключенное в круглые скобки, принимает значение, не равное нулю (ИСТИНА), выполняется инструкция1 (иногда ее называют телом цикла). Если выражение равно нулю (ЛОЖЬ), происходит выход из цикла и переход к выполнению следующей инструкции.

Тело цикла может быть составной или простой инструкцией. В последнем случае она должна заканчиваться при знаком окончания инструкции:

```
while (c == k) ans = 0;
while (priz)
```



```
{  
    p = q;  
    mys (r);  
}
```

Пример 5

```
//Программа предлагает ввод 10 чисел  
//и выводит на экран количество  
//отрицательных и положительных  
#include <stdio.h>  
void main()  
{  
    int nn = 0, np = 0, k = 1, n;  
    while (k <= 10)  
    {  
        printf("\nШаг %d. Введите целое число", k);  
        scanf ("%d", &n);  
        if (n < 0) nn = nn + 1;  
        else  
            if (n > 0) np = np + 1;  
        k = k + 1;  
    }  
    printf ("\nВведено %d отрицательных чисел", nn);  
    printf (" и %d положительных", np);  
}
```

На экран монитора выводится порядковый номер числа и подсказка для каждого из значений. После ввода всех чисел будет выведено количество отрицательных и положительных значений.

Особые операции C++

Некоторые операции языка C++ являются уникальными. Они позволяют сделать программу более эффективной и компактной. К таким операциям относятся операции уменьшения и увеличения переменной на единицу, расширенная операция присваивания и битовые операции (см. раздел «Логические операции и операции над битами»).

Операции уменьшения и увеличения переменной на единицу имеют очень простой синтаксис: имя переменной, за или перед которой для увеличения переменной на единицу вставляется оператор (++), то есть два знака «плюс», а для уменьшения переменной на единицу — оператор (--), то есть два знака «минус»:

```
++p;  
--k;  
i++;  
index--;
```

Форма записи первой пары примеров называется префиксной, второй — постфиксной. Разница между ними проявляется только в том случае, если они входят в какое-либо выражение. В префиксной форме переменная сначала увеличивается или уменьшается на единицу, затем ее новое значение подставляется в выражение. В постфиксной форме значение переменной сначала подставляется в выражение, а затем изменяется.

Увеличения числа на единицу также называют инкрементом, уменьшение — декрементом.

Пример 6

```
/*Тело цикла предыдущей программы можно сократить*/  
while (k <= 10)  
{  
    printf("\nШаг %d. Введите целое число", k++);  
    scanf ("%d", &n);  
    if (n < 0) nn++; //или ++nn;  
    else  
        if (n > 0) pr++; //или ++pr;  
}
```

В этом примере при выводе подсказки для ввода числа мы не можем использовать префиксную форму инкремента, так как значение k должно быть увеличено только после его вывода на экран.

Синтаксис расширенной операции присваивания почти не отличается от обычной: просто перед знаком равенства добавляется еще один знак операции. В программе при этом выполняются два действия: сначала с переменной из левой части инструкции и выражением из правой части выполняется операция, реализуемая первым знаком, затем полученный результат присваивается переменной:

```
r += dop; //то же, что r = r + dop;  
qwer *= k; //то же, что qwer = qwer * k;  
ap /= 50; //то же, что ap = ap / 50;
```

(См. раздел «Присваивание переменных разных типов» в главе «Перегрузка операторов», а также главу «Операторы».)

Логические операции и операции над битами

В памяти машины данные представлены в виде двоичных чисел. Один разряд такого числа называется битом. Он может принимать значение ноль (0) и единица (1). Операции над битами позволяют проверять и изменять отдельные биты переменных. Чаще всего эти операции используются в технике. Логические операции подобны операциям над битами, за исключением того, что они проверяют операнды полностью, не учитывая значения отдельных битов.

В табл. 24 приводится список операторов, определенных в C++, которые реализуют логические операции и операции над битами.

Операции, реализуемые операторами `&`, `|`, `>>`, `<<`, `&&` и `||`, являются бинарными (имеют два операнда), операции `~` и `!` — унарные (применяются к одному операнду).

Таблица 24. Операторы логических операций

Оператор	Описание
----------	----------

<code>&</code>	Поразрядное И. Устанавливает бит результата в 1, если соответствующие биты операндов установлены в 1
--------------------	--

Оператор	Описание
	Поразрядное ИЛИ. Устанавливает бит результата в 1, если один из соответствующих битов операндов установлен в 1
~	Поразрядное НЕ. Устанавливает бит результата в 1, если соответствующий бит операнда не установлен (равен 0)
>>	Поразрядный сдвиг вправо. Сдвигает все биты первого операнда вправо на число разрядов, заданное вторым операндом
<<	Поразрядный сдвиг влево. Сдвигает все биты первого операнда влево на число разрядов, заданное вторым операндом
&&	Логическое И. Устанавливает результат в истину (1), если оба операнда истинны (1). В противном случае результат — ложь (0)
	Логическое ИЛИ. Устанавливает результат в истину (1), если один из операндов — истина (1). В противном случае результат — ложь (0)
!	Логическое НЕ. Устанавливает результат в истину (1), если операнд — ложь (0). В противном случае результат — истина (1)

Пример 7

```
a = 5;  
c = a & 1; //c равно 1;  
c = a && 1; //c равно 0;
```

Операции над полями бит

С и С++ предоставляют очень удобное средство для работы с битами: поля бит. Структура данных, использующая поля бит, объявляется следующим образом:


```
struct [имя_структуры]
{
    unsigned поле1:размер1;
    unsigned поле2:размер2;
    ...
    unsigned полеN:размерN;
} [переменные_структуры];
```



ПРИМЕЧАНИЕ Максимальное число, размещаемое в поле длиной n бит, равно 2 в степени $n-1$.

Допустим, тройка чисел a , b и c определяет некоторую дату: переменная a может принимать значения от 1 до 31 (число), переменная b — от 1 до 12 (месяц), переменная c — от 1 до 7 (день недели). Можно объявить их как обычные целые числа, однако при большом количестве дат это потребует большого объема памяти. Значительно рациональнее разместить эти числа в структуре и каждому из них поставить в соответствие свое поле бит. При этом для переменной a потребуется шесть бит, для b — пять бит и для c — четыре бита, таким образом, вся информация о дате разместится в одном слове памяти (двух байтах).

```
struct MyDate
{
    unsigned day:6;
    unsigned month:5;
    unsigned day_of_week:4;
};
```

Поля бит размещаются в слове слева направо: в приведенном примере правые 4 бита займет информация о дне недели, следующие 5 бит влево — информация о месяце, далее расположатся 6 бит числа, крайний левый бит останется свободным.

Указатели, массивы и строки

Использование в явном виде указателей и связанной с ними косвенной адресации является достаточно сложной темой для начинающих программистов, но владение этим методом делает программирование на C++ более эффективным.

В этой главе также вводятся новые типы: массивы данных, символы и строки, описываются различные операторы динамического распределения памяти.

Указатели и передача данных

Работа с указателями является необходимым атрибутом любого языка, однако иногда она выполняется в неявном виде. В C++ программист имеет возможность контролировать этот процесс и вносить в него свои коррективы.

Попросту говоря, указатель — это переменная или параметр, в котором располагается адрес некоторого фрагмента данных. Указатель обычно занимает в памяти компьютера два или четыре байта. Преимущество указателя в том, что при его использовании можно обращаться к фрагменту данных любого размера.

Предположим, две функции должны обрабатывать одни и те же данные, например, последовательность символов для вывода на экран монитора. Можно, конечно, объявить их глобальными. Однако, как уже говорилось выше, большое количество глобальных переменных приводит к нерациональному использованию оперативной памяти компьютера и является признаком плохого стиля программирования.

Если мы объявляем данные локальными, для обмена ими между функциями можно использовать один из двух способов:

- передача значений данных. Этот метод подходит в том случае, если данные занимают мало места в памяти компьютера, но становится неэффективным, если обрабатываются массивы, строки и другие большие объекты;
- передача адреса данных, который занимает столько же места, сколько целое число, но сообщает процессору, где найти нужные данные. Способ обращения к данным по их адресу называется *косвенной адресацией*.

Следует заметить, что обмен данными между функциями осуществляется через *стек* — область памяти компьютера, специально выделяемую для этой цели. При вызове какой-либо функции копии значений фактических параметров записываются в стек в соответствии с типами, указанными в ее прототипе. Затем происходит переход в вызываемую функцию. Она извлекает значения из стека в соответствии с типами, указанными в заголовке определения функции, обрабатывает их и с помощью инструкции `return` помещает в стек результат работы функции. Наконец, результат извлекается из стека и возвращается в вызывающую функцию.

Таким образом, при передаче данных напрямую (как в первом случае) создается их полная копия и организуется доступ ко всем данным. Однако изменение информации происходит только в скопированном фрагменте, а оригинал данных остается без изменения. При выходе из функции оригинал данных также не изменяется.

При передаче адреса данных (как во втором случае) в стеке размещается только два или четыре байта копии адреса информации. С ее помощью вызываемая функция может не только использовать данные, но и изменять их. При выходе из функции данные сохраняют измененные значения. Следовательно, можно отметить два преимущества второго способа передачи данных: эффективное использование стека при работе с фрагментами данных любого размера и возможность изменения информации.



ПРИМЕЧАНИЕ Эти два способа работы с параметрами функций используются во многих языках, иногда их называют передачей параметров *по значению* (by value) и *по ссылке* (by reference).

Передача данных по ссылке

Для передачи данных по ссылке следует выполнить следующие шаги.

1. В списке аргументов прототипа и заголовка функции объявить параметры, используя для этого оператор косвенной адресации (*).
2. Убедиться, что передача параметров осуществляется по адресу. Как правило, это достигается использованием оператора определения адреса (&).
3. Для получения значения параметра, который является указателем, к нему применяется оператор косвенной адресации (*).

Предположим, нам надо передать функции myfunc параметр par1 целого типа по ссылке. В вызове функции используется переменная i целого типа. Тогда

```
void myfunc (int *par1); //прототип функции
myfunc (&i); /* вызов функции, значение i после этого
увеличится на 10 (см. тело функции)*/
void myfunc (int *par1) //заголовок функции
{*par1 += 10;} //тело функции
```

Запись *par1 означает значение, находящееся по адресу par1. Соответственно, описание int *par1 можно прочитать так: значение, находящееся по адресу par1, является целым. Использование par1 вместо *par1 в теле функции будет означать увеличение на 10 адреса, заданного переменной par1, его значение (*par1), находящееся по этому адресу, не изменится.



ВНИМАНИЕ Не забывайте использовать оператор косвенной адресации (*), чтобы указать значение параметра, передаваемого по ссылке, и оператор определения адреса (&) в вызове функции с таким типом передачи.

Пример 8

```
//Программа выводит на экран куб
//введенного числа
#include <stdio.h>
void p3 (float *num);
void main ()
{
    float wf;
    printf ("\nВведите значение числа:");
    scanf ("%f", &wf);
    p3 (&wf); //wf присвоить wf в кубе
    printf ("Куб числа равен %7.2f.", wf);
}
void p3 (float *num) //функция вычисления куба
{
    //умножаем num само на себя 3 раза
    *num = *num * *num * *num;
    /* или *num *= *num * *num; */
}
```

Функцию main можно записать по-другому:

```
void main ()
{
    float *wf;
    printf ("\nВведите значение числа:");
    scanf ("%f", wf);
    p3 (wf); //wf присвоить wf в кубе
    printf ("Куб числа равен %7.2f.", *wf);
}
```

Результат работы программы будет тем же.

Одномерные массивы

Понятие *массивов* используется практически во всех языках программирования в силу двух основных причин:

- они обеспечивают удобный способ для размещения данных большого размера в памяти программы;
- их можно использовать для выполнения огромного количества операций с помощью нескольких строк программного кода, оформленного в виде инструкций цикла.

Массив — это некоторая последовательность чисел в памяти компьютера. Синтаксис описания одномерного массива выглядит так: тип элемента массива, один или несколько пробелов, имя массива и его длина (число элементов), заключенная в квадратные скобки.

```
float myarray[135];  
long farr[44];  
int work[N];
```

В последнем примере число N должно быть определено до его использования в инструкции описания. Это можно сделать, выполнив директиву `#define N` константа (см. также главу «Директивы, макросы и операторы препроцессора») или просто подставив целое число вместо N.



ВНИМАНИЕ К моменту описания массива число его элементов должно быть фиксировано: это может быть константа или константное выражение.

В памяти компьютера элементы массива располагаются последовательно. Каждый элемент занимает одно и то же количество байт в соответствии с указанным типом. Нумерация элементов начинается с 0, последний элемент имеет индекс (номер), равный длине массива минус один. Общий объем памяти, занимаемой массивом, определяется как произведение длины его элемента на количество элементов. Доступ к элементу массива осуществляется по его индексу и имени массива. Например, первый элемент массива `myarray` записывается как `myarray[0]`, последний — как `myarray[134]`.

Многомерные массивы и их размещение в памяти компьютера

Чтобы разместить в памяти компьютера таблицу чисел, удобнее использовать двухмерный массив. Описание двухмерного массива отличается от описания одномерного только наличием еще одного значения длины в квадратных скобках. Первое значение — количество строк в таблице,

второе — количество столбцов. В памяти компьютера элементы таблицы располагаются построчно: сначала первая строка, затем вторая и т. д. Каждый элемент таблицы имеет два индекса: номер столбца и номер строки, в которых он находится. Все индексы начинаются с нуля.

Пример 9

```
unsigned a[3][4];  
a[0][0] = 0; //1-й элемент таблицы  
a[0][i] = i; //i-й элемент первой строки  
a[j][i] = 4*j + i; //i-й элемент j-й строки  
//i изменяется от 0 до 3, j от 0 до 2
```

В памяти элементы располагаются следующим образом: $a[0][0]$, $a[0][1]$, $a[0][2]$, $a[0][3]$, $a[1][0]$, $a[1][1]$, ..., $a[1][3]$, $a[2][0]$, ..., $a[2][3]$.

Каждый следующий элемент больше предыдущего на единицу.

Если имеется несколько одинаковых таблиц, их лучше организовать в трехмерный массив. При этом следует учесть, что первым индексом элемента будет номер таблицы, вторым — номер строки, третьим — номер столбца.

```
unsigned tables[5][3][4]; //5 таблиц размером 3x4  
tables[k][j][i];  
//i-й элемент j-й строки из k-й таблицы
```

В памяти последовательно разместятся таблицы с нулевой по четвертую. Алгоритм распределения памяти для каждой таблицы описан в примере 9.



ПРИМЕЧАНИЕ При последовательной работе со всеми элементами массива чем правее стоит индекс, тем быстрее он изменяется. Говоря иначе, элементы многомерных массивов хранятся в памяти в порядке возрастания самого правого индекса.

Процесс увеличения размерности массива (количества индексов, необходимых для определения места элемента в нем) можно продолжить. Размерность массива в C++ формально не ограничивается, однако следует иметь в виду,

что общий объем памяти, занимаемой массивом, определяется как произведение его длин по каждому из индексов на длину элемента. Таким образом, трехмерный массив чисел с плавающей точкой `float tables[16][128][256]` занимает $16 \cdot 64 \cdot 256 \cdot 4 = 1\,048\,576$ байт, то есть 1 Мбайт оперативной памяти. В то же время, с помощью стандартных функций в DOS можно получить доступ к памяти не более 640 Кбайт, причем из этого числа надо вычесть часть, уже занятую самой системой DOS, драйверами (например, драйверами для устройства чтения компакт-дисков и русификатора), средой программирования и собственно программой). Использование такого массива без специальных средств в среде DOS невозможно.



ВНИМАНИЕ Изменение индекса массива от нуля до его длины минус 1 требует тщательного определения начального и конечного условий выполнения циклов. Компилятор C++ не выполняет проверку индексов массивов.

Пример 10

```
/*Определение в каждой строке таблицы из M строк и N
  столбцов минимального значения элемента, вывод его на
  экран монитора, M и N – константы*/
#include <stdio.h>
#define M 3
#define N 4
void main ()
{
    int i, j, min, marks[M][N];
    // ввод элементов таблицы
    j = 0;
    while (j < M)
    {
        i = 0;
        while (i < N)
        {
            printf("Введите элемент [%d][%d]", j, i);
            scanf ("%d", &marks[j][i]);
        }
    }
}
```


Пример 10**(продолжение)**

```
        j++;
    }
//основная часть
    j = 0;
    while (j < M)
    {
        i = 0;    min = marks[j][0];
        while (i < N)
        {
            if(marks[j][i]<min) min=marks[j][i];
            i++;
        }
        printf("\nМинимальный элемент строки %d равен %d",
                j, min);
        j++;
    }
}
```

После запуска программы по подсказкам нужно ввести каждый из элементов массива, после этого программа определит минимальное значение каждой строки и выведет его на экран.

Инициализация массивов

В примере 10 элементы массива `marks` вводились с клавиатуры. Можно задавать их значения с помощью оператора присваивания (например, `M=2, N=3`).

```
marks[0][0]=6; marks[0][1]=-7; marks[0][2]=15;
marks[1][0]=3; marks[1][1]=7; marks[1][2]=-5;
```

Однако в C++ есть и более компактный способ задания начальных значений элементов массива: при его описании (аналогичный способ инициализации констант уже описывался).

Инициализацию массива `marks` можно сделать так:

```
int marks[2][3] = {6, -7, 15, 3, 7, -5};
```




ПРИМЕЧАНИЕ Значение самого левого индекса можно опустить: компилятор сам определит его по количеству элементов.

В C++ допускается частичная инициализация первых элементов массива. Допустим, надо инициализировать только 1-й элемент 1-й строки таблицы `marks` и 1-й элемент 2-й строки.

```
int marks[][3] = {{6}, {3}};
```

Для массивов, элементы которых можно задать с помощью того или иного алгоритма, используют цикл. Это имеет особенно важное значение для больших массивов.

Пример 11

```
int marks[2][3], i, j;  
i = 0;  
while (i < 2)  
{  
    j = 0;  
    while (j < 3) marks[i][j++] = i * 3 + j + 1;  
    i++;  
    //В памяти массив расположится так: 1, 2, 3, 4, 5, 6  
}
```

Массивы и указатели

В C++ указатели имеют тесную связь с массивами, так как при использовании указателей обработка массивов выполняется значительно эффективнее, особенно для многомерных массивов.

Следует отметить, что в программе адрес массива обозначается так же, как его имя. Так, `marks` — адрес массива `marks`. Учитывая, что адрес массива совпадает с адресом его первого элемента, адрес массива можно записать по-другому: `&marks[0]`.

Помимо первого способа задания массива, который использовался в примере 11, можно задать массив еще двумя способами.


```
//способ 2
int marks[2][3], i;
i = 0;
while (i < 2*3)
    *((int *)marks + i++) = i+1;
//способ 3
int marks[2][3], *pmarks=(int *)marks, i;
i = 0;
while (i < 2*3)
    *pmarks++ = ++i;
```

Во всех случаях начальные значения элементов массива marks будут одними и теми же.

Нужно обратить особое внимание на необходимость приведения типов в четвертой выполняемой строке второго способа и в первой выполняемой строке третьего способа в связи с более строгими по сравнению с С требованиями C++ к совпадению типов указателей.



СОВЕТ Полезно называть указатель на переменную или массив именем этой переменной или массива с буквой p перед ним (p — pointer, указатель). Например, `int param, *pparam;`. Далее можно определить указатель pparam на целое число param с помощью инструкций `pparam=&param;` или `*pparam=param;`.

Итак, элементы массивов могут определяться двумя способами: с помощью индексов и с помощью указателя. Если таблица table описана как `int table[20][15]`, то запись `*(table+5)` означает то же, что `table[0][5]`, а `*(table+35)` — то же, что `table[2][5]`. (См. также раздел «Многомерные массивы и их размещение в памяти компьютера».)

Строки

В C++ определен еще один тип — символьный (`char`). Он является разновидностью целого числа (см. также главу «Типы данных») и может принимать значения от -128 до 127 с учетом знака (`char`) или от 0 до 255 без учета знака (`unsigned char`). Обычно символ занимает в памяти один

байт и для персональных компьютеров является ASCII-кодом (см. также приложение «Таблица ASCII-кодов»).

Символьная константа — это символ, заключенный в одинарные кавычки. *Управляющим символом* называют последовательность, начинающуюся с символа обратный слэш (\), которая также заключается в одинарные кавычки и занимает в памяти компьютера 1 байт.

```
char ch1 = 'a', ch2 = 'A', wch;  
char STREND = '\n';
```

Строка определяется как массив (последовательность) любых символов, заключенных в двойные кавычки. Признаком конца строки является управляющий символ ноль-терминатор \0. Такой массив называют строкой в формате ASCIIZ или ASCIIZ-строкой.



ВНИМАНИЕ Управляющие символы \n и \0 — разные символы. Ноль-терминатор — признак конца строки, и используется при работе с ASCIIZ-строками; \n — признак перевода строки и, используется при выводе.

Если в процессе выполнения программы строка может принимать несколько значений, то в ее описании должна быть указана максимально возможная длина плюс один (для ноль-терминатора). Объем памяти, выделенный для строки, остается неизменным в области ее видимости, но все символы, стоящие после признака конца строки, игнорируются.

Допустим, строка описана как массив из 80 символов, но 15-й элемент равен \0. При ее выводе с помощью функции printf на экране появится 15 символов (с 0-го по 14-й элемент массива).

Строка символов, как и любой другой массив, может быть инициализирована при ее описании.

```
char wstr[50], start_str[] = "START";  
char wstr1[20] = ""; //пустая строка  
char prompt[7] = {'Ч', 'И', 'С', 'Л', 'О', ' ', '\0'};
```




СОВЕТ Инициализируя строку, не изменяемую во время работы программы, проще опустить ее длину. При задании, например, `char error_str[] = "Error"`, компилятор отведет для строки в памяти 6 байт. Однако если строка принимает несколько значений, лучше определить ее максимально возможную длину заранее (не забыв увеличить на единицу).



ВНИМАНИЕ Так как строковые переменные в C++ являются массивами, оператор равенства при работе с ними может быть использован только для инициализации.

Во всех остальных случаях: при копировании, присваивании, конкатенации, то есть объединении, и т. д. — операции со строками должны выполняться с помощью функций или инструкций цикла для каждого символа строки.

Например, для присваивания строке начального значения в процессе выполнения программы, а не при ее объявлении, следует использовать библиотечную функцию `strcpy`:

```
strcpy (firststr, "START");
```

Заголовки функций, используемых для работы со строками, находятся в файле `string.h` и включаются в программу с помощью директивы `#include` (см. также главу «Директивы, макросы и операторы препроцессора»).

Таблица 25. Некоторые функции работы со строками

Функция	Описание
<code>strlen (char *s)</code>	Возвращает число символов до нуль-терминатора
<code>strncpy (char *dest, char *src, int n)</code>	Копирует максимум n символов из строки src в строку dest
<code>strcat (char *dest, char *src)</code>	Дописывает содержимое src в конец dest
<code>strcmp(char *str1, char *str2)</code>	Посимвольно сравнивает две строки

Некоторые другие функции для работы со строками приведены в табл. 25 (см. также раздел «Функции работы со строками» в главе «Библиотечные функции»).

Пример 12

```
#include <stdio.h>
#include <string.h>
void main()
{
    char *prompts[] = {"фамилия", "имя", "отчество"};
    char name[20], fio[60];
    int i=0;
    strcpy (fio, ""); //запись в fio пустой строки
    while (i < 3)
    {
        printf ("\nВведите %s:", prompts[i]);
        scanf ("%s", name);
        strcat (fio, name);
        if (i++ < 2) strcat (fio, " ");
    }
    printf ("ФИО клиента: %s", fio);
}
```

Программа принимает с клавиатуры фамилию, имя, отчество, записывает их в виде одной строки, разделяя пробелами, и выводит полученную строку на экран монитора.

Спецификатор %s используется для вывода и ввода данных, имеющих тип указателя (адреса) строки.

Практически все адреса в этой программе являются адресами строк, то есть адресами символьных массивов, и поэтому совпадают с их именами.

Указатели и динамическое распределение памяти

До сих пор мы говорили о данных, для которых память компьютера выделялась при объявлении переменных. Такой способ распределения памяти называется *статическим*.

Однако иногда размер данных становится известным только во время выполнения программы. Например, если в процессе какого-либо измерения выполняется сохранение данных через определенные промежутки времени, объем этих данных зависит от времени, прошедшего с начала измерения.

В таком случае рациональнее распределять память компьютера во время измерения. Процедура выделения памяти во время выполнения программы называется *динамическим распределением* (выделением) памяти.

В C++ существует два способа динамического выделения памяти. Один из них, унаследованный от C, использует стандартные библиотечные функции `malloc` и `free`. Другой — операторы `new` и `delete`, которые отсутствуют в C (см. раздел «Использование операторов `new` и `delete`»).



ПРИМЕЧАНИЕ Для обоих способов необходимо применение переменных типа указателя. Как правило, конкретные адреса, содержащиеся в этих переменных, не используются программистом.

Использование стандартных функций `malloc` и `free`

Динамическое распределение памяти с помощью библиотечной функции `malloc` состоит из следующих шагов.

1. Включение в программу файла заголовков `malloc.h` директивой `#include <malloc.h>`.
2. Объявление указателя нужного типа, например `int *p;`.
3. Вызов функции `malloc` с указанием в качестве параметра требуемого количества памяти в байтах. Так как функция выдает результат своей работы в виде указателя на тип `void`, выполняем приведение типа (преобразуем тип результата к типу, указанному в объявлении). Присваиваем полученное значение объявленному указателю: `p=(int *) malloc (число элементов*sizeof(int));`. Вместо `int` может быть подставлен любой стандартный или введенный программистом тип.

4. Проверка факта выделения памяти. Если выделение памяти выполнено, продолжаем выполнение программы, если нет, выходим из нее с соответствующей диагностикой о недостатке памяти: `if (p) продолжение; else выход; .` Пояснения к этому шагу приводятся ниже.
5. Освобождение памяти после окончания работы с ней. Для этого вызываем функцию `free` и используем указатель в качестве аргумента: `free (p); .`



ВНИМАНИЕ Значение указателя, полученное после выполнения шага 3, должно сохраняться до выполнения шага 5. В противном случае вся память, выделенная по этому адресу, будет потеряна при выходе из программы, что, в конечном счете, может привести к недостатку памяти и нарушению работы операционной системы.

Если выделение памяти в нужном объеме невозможно, функция `malloc` возвращает в качестве своего результата нулевой указатель `NULL`, соответствующий значению `ложь`.

Для проверки доступного объема свободной памяти можно использовать функцию `long unsigned coreleft()`. Ее вызов также может быть полезен для проверки корректности выделения памяти в программе (значения функции при входе и выходе должны совпадать).



ВНИМАНИЕ Наиболее частой причиной «зависания» компьютера при работе с динамически выделяемой памятью является несоответствие инструкций `malloc` и `free` или недостаточный объем свободной памяти.

Пример 13

```
#include <malloc.h>
#include <stdio.h>
void main()
{
    int i=0, num;
    float *table;
    printf ("\nВведите длину таблицы:");
```


Пример 13**(продолжение)**

```
scanf ("%d", &num);
table=(float *)malloc(num*sizeof(float));
if (table == NULL) //или if (!table)
{
    printf("\nНедостаточно памяти %ld", coreleft());
    return;
}
while (i < num)
{
    printf ("\nВведите %d-й элемент:", i);
    scanf ("%f", &table[i++]);
    // или scanf ("%f", table + i++);
}
printf ("\nПредпоследний элемент:%7.2f", table[num-
2]);
free (table);
}
```

Программа запрашивает ввод длины таблицы, ее элементов и выводит на экран предпоследний элемент.

Спецификатор %ld является спецификатором формата, необходимым для вывода длинного целого числа (long int, или просто long).

Для динамического выделения памяти можно также использовать функцию calloc().



ПРИМЕЧАНИЕ В зависимости от используемой версии C++ для работы с большими фрагментами динамической памяти возможно применение функций farmalloc(), farcalloc(), farcoreleft() и farfree().

Использование операторов new и delete

Операторы new и delete могут быть использованы везде, где используются функции malloc и free. Однако этот способ имеет несколько преимуществ:

- не нужно включать в программу файл malloc.h;

- не нужно выполнять приведение типов — это сделает оператор new;
- самое главное, эти операторы не просто выделяют и освобождают память. Когда оператор new используется для размещения объекта в памяти, он автоматически вызывает конструктор объекта, если таковой существует. Аналогично, при освобождении памяти с помощью оператора delete последний вызывает деструктор объекта (также при его наличии) (см. также раздел «Конструкторы и деструкторы: рождение и смерть объектов» в главе «Классы»).

Выделение памяти с помощью оператора new имеет следующий синтаксис:

указатель = new тип;

указатель = new тип [число элементов];

Вторая строка описывает синтаксис оператора при выделении памяти для массивов.

Так же как функция malloc, оператор new возвращает значение NULL, если объем требуемой памяти больше, чем объем свободной. Рекомендуется проверить полученное значение указателя прежде, чем продолжать выполнение программы.

Освобождение памяти с помощью оператора delete производится только в том случае, если она была выделена с использованием оператора new и имеет следующий синтаксис:

delete указатель;

delete [число элементов] указатель;



ВНИМАНИЕ Использование квадратных скобок для освобождения памяти, выделенной под массив, обязательно. Как и при инициализации, его длину можно опустить, но скобки опускать нельзя, иначе освободится память, занимаемая одним элементом.

Перепишем пример 13 с применением операторов new и delete:

Пример 14

```
#include <stdio.h>
void main()
{
    int i=0, num;
    float *table;
    printf ("\nВведите длину таблицы:");
    scanf ("%d", &num);
    table = new float [num];
    if (!table)
    {
        printf("\nНедостаточно памяти");
        return;
    }
    while (i < num)
    {
        printf("\nВведите %d-й элемент:", i+1);
        scanf ("%f", table + i++);
    }
    printf ("\nПредпоследний элемент:%7.2f",
            table[num-2]);
    delete [] table;
}
```


Ввод и вывод

До сих пор использовались простейшие способы ввода с клавиатуры и вывода на экран монитора с помощью библиотечных функций `printf` и `scanf`. В этой главе для расширения возможностей ввода и вывода информации вводятся дополнительные функции из библиотеки `stdio.h`. Помимо них предлагается объектно-ориентированный метод ввода / вывода с помощью стандартных потоков `cin` и `cout`.

В главе также описывается работа с файлами с использованием библиотечных функций из файла `stdio.h` и стандартных операторов потоков для работы с файлами `inf` и `outf`.

Понятие потока

Понятие *потока* (`stream`), используемое в программировании, тесно связано с обычным, бытовым пониманием этого слова. *Поток ввода* можно сравнить с трубой, по которой вода (информация) поступает в бассейн (память компьютера), *поток вывода* — с трубой, по которой вода выходит из бассейна. Важной особенностью этой трубы является то, что в каждый момент времени данные могут двигаться только в одном направлении. Даже если одна и та же труба используется для ввода и вывода, это не может происходить одновременно: для переключения направления потока его нужно остановить, выполнить некие действия и лишь затем направить поток в обратном направлении. Другой особенностью потока является то, что он почти никогда не иссыкает. Иногда он высыхает, но этот период не может быть долгим, если система функционирует нормально.

Файлы также легко представляются в виде потоков. При выполнении ввода или вывода всегда можно получить или вывести очередной байт при условии, что система не находится в критической ситуации.

Массивы также можно трактовать как потоки. Однако следует иметь в виду, что в отличие от вывода файла их размер строго ограничен.

Главным свойством потоков в C++ является возможность получения или вывода очередного байта или группы байтов.

Операторы потоков для ввода/вывода

Операторы ввода (>>) и вывода (<<) для работы с потоками являются альтернативой функциям `printf`, `scanf` и другим, определенным в файле `stdio.h`. Преимущество этих операторов заключается в следующем:

- отпадает необходимость использования спецификаторов формата, если формат стандартного вывода, установленный по умолчанию, обеспечивает потребности программиста;
- их можно расширить для работы со своими классами. (См. также главу «Объекты и классы библиотеки ввода/вывода».)



ПРИМЕЧАНИЕ В стандартной библиотеке C++ для работы с функциями ввода/вывода, определенными в файлах `stdio.h` и `iostream.h`, реализованы разные буферы. Поэтому их одновременное использование в программе может вызвать непредсказуемые последствия. При невозможности избежать смешивания этих функций с потоками ввода/вывода следует использовать функцию `sync_with_stdio()` (см. также раздел «Объекты и классы библиотеки ввода/вывода»).

Пример 15

```
//Программа вычисления длины окружности
//по ее радиусу
#include <iostream.h>
void main()
{
    int r;
    float pi = 3.14;
    cout << "\nВведите значение радиуса:";
```



```
cin >> r;  
cout << "Длина окружности с радиусом ";  
cout << r;  
cout << "см равна ";  
cout << 2 * pi * r;  
cout << "см.";
```

При работе с потоками ввода/вывода следует учесть несколько важных моментов:

- в программу должен быть включен файл `iostream.h`. Файл `stdio.h` при этом исключается;
- не нужно использовать спецификаторы формата — формат вывода объекта определяется его типом;
- нет необходимости использовать оператор определения адреса (`&`);
- данные выводятся в стандартный поток вывода (`cout`), который обычно подразумевает экран монитора `cout << "Введите значение радиуса:"`;;
- данные вводятся в стандартный поток ввода (`cin`), который обычно подразумевает клавиатуру `cin >> r`;;
- в потоках для перехода на новую строку используется символ `'\n'` (в некоторых версиях C++ можно использовать также `endl`).

Символы `>>` и `<<` могут показаться непривычными, но их значение становится более понятным, если проследить направление потоков (передачи) данных.

 **ПРИМЕЧАНИЕ** Операторы потоков ввода/вывода похожи на логические операторы сдвига (см. также табл. 24 в главе «Основные особенности программирования на C++»), и это не случайно. Использование операторов сдвига влево и вправо является примером перегрузки (*overloading*) операторов в C++ (см. также главу «Перегрузка операторов»). Эти операторы перегружены для классов `istream` и `ostream`, определенных в библиотеке `iostream.h`, и действия, выполняемые ими, полностью изменились.

Последние пять строк вывода из примера 15 можно записать короче:

```
cout << "Длина окружности с радиусом " << r << "см равна " <<  
2 * pi * r << "см.";
```

В главе «Основные особенности программирования на C++» подробно описан механизм выполнения присваивания одного и того же значения четырем переменным. Подобным образом происходит и выполнение последнего оператора вывода:

- выполняется вывод первой части строки: `cout << "Длина окружности с радиусом "`. Ее результатом является `cout`;
- в полученный результат выводится следующая часть строки: `cout << r`. Результат — опять `cout`;
- в полученный результат выводится очередная часть строки: `cout << "см равна "`.

Аналогичным образом можно ввести несколько переменных в одной строке: `cin >> table[0][0] >> table[0][1] >> table[0][2] >> table[1][0];`.

В этом примере четыре элемента таблицы вводятся одной строкой.

Как и в предыдущем случае, результатом ввода `cin >> table[0][0]` является `cin`, затем выполняется `cin >> table[0][1]` и т. д.

Управление форматом при работе с потоками

Обычно программисту не нужно задумываться о том, какой формат будет применен при выводе его числа: по умолчанию устанавливается десятичный формат, являющийся самым распространенным. Но что делать, если нужно вывести число в шестнадцатеричной системе счисления или вывести ASCII-код символа? Для этого используются специальные виды объектов, которые иногда называют флагами форматирования (см. также главу «Объекты и классы библиотеки ввода/вывода»).

Пример 16

```
int num; char wch='A';
cout<<"num="<<hex<<num<<"(16)"<<dec<<num<<"(10)";
/*вывод числа num сначала в 16-ричной, затем в
  десятичной системе счисления*/
cout<<"ASCII-код символа "<<wch<<"="<<(int)wch;
```

Следует заметить, что установленный формат действует на весь последующий вывод, и для восстановления старого формата нужно явно задать его через поток вывода. Для восстановления десятичной системы счисления при выводе используется флаг `dec`.

Ввод/вывод строк с помощью библиотеки `stdio.h`

Некоторые программисты на C++ предпочитают использовать для ввода/вывода библиотечные функции, объявленные в файле `stdio.h` и унаследованные от C. Это оправдано, если другие библиотеки, которые требуются для программы, тоже написаны на C.

Библиотечные функции, объявленные в файле `stdio.h`, включают не только функции `printf` и `scanf`, но и еще ряд функций. Среди них функции, ориентированные на ввод строки `puts` и ее вывод `gets`.

Использование инструкций `printf("%s\n", str);` и `puts(str);` приводит к одному и тому же результату. Последний вариант, однако, более эффективный, так как позволяет избежать процедуры форматирования. В то же время, он менее гибкий из-за того, что выполняет переход на новую строку независимо от того, нужно это или нет.

Одной из наиболее полезных функций для ввода строк является функция `gets()`. Эта функция принимает все символы, набранные на клавиатуре до нажатия клавиши `Enter` (в том числе пробелы), и размещает их в строке, адрес которой указывается в качестве параметра функции. Далее содержимое строки можно интерпретировать как угодно, включая определение разделителей, пробелов, цифр и т. д.



ВНИМАНИЕ Функция `gets()` позволяет вводить пустые и содержащие пробелы строки. Функция `scanf` и оператор ввода `>>` не поддерживают возможность ввода пробелов: они воспринимают пробел как разделитель и признак конца ввода.

Пример 17

/*Программа предлагает ввод строки с клавиатуры, считает количество слов и формирует из них массив `text`.

Разделителями считаются пробелы, знаки препинания и конец строки (перенос слов в тексте не допускается)*/

```
#include <stdio.h>
void main()
{
    char str[81], text[40][81];
    char *p=str, *s;
    int wnum=0, neww=0, i=0;
    puts ("\nВведите строку: ");
    gets(str);
    s = text[i++];
    while (*p!='\0')
    {
        if(*p!=' ' && *p!=',' && *p!='.' && *p!=':' && *p!=';' && *p!='!' && *p!='?' && *p!='-' )
        {
            neww=1;
            *s++ =*p;
        }
        else
        {
            if (neww)
            {
                wnum++;
                *s='\0';
                s = text[i++];
            }
            neww = 0;
        }
    }
}
```



```
        *p++;
    }
    if (neww) wnum++;
    if (wnum > 0) s = text[i++];
    *s = '\0';
    //построчный вывод массива text
    printf ("Число слов в строке %d\n",wnum);
    i=0; s=text[0];
    while(*s!='\0')
    {
        s=text[i++];
        puts(s);
    }
}
```

Функции для работы с файлами в библиотеке stdio.h

Файловый ввод/вывод в C++ несложен и при использовании стандартных функций, объявленных в библиотеке stdio.h, состоит из пяти основных шагов.

1. Включение файла заголовков stdio.h и объявление указателей файлов FILE *f1, *f2, *fk;.
2. Открытие файла вызовом функции fopen, присвоение возвращаемого ею значения указателю файла. Параметрами функции являются две строки, одна из которых — имя файла, другая — тип доступа к нему.
3. Проверка правильности открытия файла, то есть указатель файла после шага 2 не должен равняться NULL.
4. Чтение из файла и/или запись в файл с помощью функций fprintf, fscanf, fputs, fgets и других, объявленных в библиотеке stdio.h. Они аналогичны функциям printf, scanf, puts, gets, за исключением того, что имеют еще один параметр — указатель файла.
5. Заккрытие файла вызовом функции fclose с указателем файла в качестве параметра.



ВНИМАНИЕ Если имя файла включает в себя имена папок, в которых он находится, то они отделяются друг от друга и от имени файла двойным обратным слэшем. Например, c:\\mydir\\dir1\\myfile.txt или dir1\\myfile.txt.

Пример 18

```
#include <stdio.h>
void main()
{
    FILE *f1;
    char wstr[20];
    f1 = fopen ("d:\\tests\\mytest.txt", "w");
    if (f1 != NULL)
    {
        fputs (f1, "Привет!");
        fclose (f1);
    }
    else puts ("Ошибка открытия файла для записи.");
    f1 = fopen ("d:\\tests\\mytest.txt", "r");
    if (f1 != NULL)
    {
        fgets (f1, wstr);
        puts ("Прочитана строка:");
        puts (wstr);
        fclose (f1);
    }
    else puts ("Ошибка открытия файла для чтения.");
}
```

Программа открывает новый файл для записи (или стирает содержимое старого) и записывает в него одно слово: «Привет!». Файл закрывается и снова открывается, уже для чтения. На экран выводятся две строки:

Прочитана строка:
Привет!

При невозможности открытия файла выводится соответствующая информация.

Возможно открытие файла для чтения и записи одновременно. В этом случае вторым параметром функции `fopen` должна быть строка `"rw"` или `"wr"`.



ВНИМАНИЕ Вывод в файл производится через буфер, содержимое которого записывается в файл только после закрытия последнего или завершения программы. В связи с этим правильно прочитать новое содержимое записанного файла можно только после выполнения функции `fclose` (см. главу «Библиотечные функции»).

Имя файла может быть введено с клавиатуры. Для этого в программе должны быть строки:

```
FILE *f1;  
char filename[40];  
printf ("\nВведите имя файла:");  
gets (filename);  
if ((f1 = fopen (filename, "w")) == NULL)  
    printf ("Ошибка открытия файла %s", filename);
```

Все файлы в этом разделе были текстовыми, то есть файлами с последовательным доступом. При таком доступе чтение файла во многом подобно вводу с клавиатуры, а запись в файл — выводу на экран.

Так же просто выполняется чтение/запись в двоичных файлах прямого доступа. Дополнительным преимуществом этих файлов является возможность доступа к любому элементу файла с помощью перемещения указателя файла (см. также главу «Объекты и классы библиотеки ввода/вывода»).

Операторы потоков для работы с файлами

Понятие потока может быть применено к файлам, находящимся на накопителях, так же, как к выводу данных непосредственно на экран или их вводу с клавиатуры. C++ поддерживает объекты типа `ifstream` (input file stream, поток вводимого файла) и `ofstream` (outputstream, поток выводимого файла). Эти объекты создаются при определении имени файла во время инициализации.

Таким образом, работа с файлами с использованием операторов потоков значительно упрощается и состоит всего из трех шагов.

1. Включение файла заголовков `fstream.h`, который автоматически включает и все содержимое файла `iostream.h`.
2. Создание объектов файловых потоков с определением для них имен файлов: `ifstream inf (<имя файла >);` — для ввода, `ofstream outf (<имя файла>);` — для вывода. Имя файла должно иметь строковый тип. Имена потоков `inf` и `outf` могут быть заменены любыми другими именами переменных.
3. Работа с самими потоками записи в файл (`outf << <данные>`) и чтения из файла (`inf >> <переменная>`).



ПРИМЕЧАНИЕ Если имя файла включает в себя имена папок, в которых он находится, то они отделяются друг от друга и от имени файла двойным обратным слэшем. Например, `c:\\mydir\\dir1\\myfile.txt`.

Пример 19

```
#include <fstream.h>
void main()
{
    ofstream outf1 ("d:\\tests\\mytest.txt");
    if (outf1 == NULL)
    {
        cout<<"Ошибка открытия файла для записи.";
        return;
    }
    outf1 << "Привет!"; //вывод строки в файл
    outf1 <<flush; //закрытие файла
    if (inf1 != NULL)
    {
        inf1 >> wstr;
        cout<<"Прочитана строка:"<<'\\n'<< wstr<<'\\n';
    }
    else cout <<"Ошибка открытия файла для чтения.";
}
```


Эта программа выполняет те же действия, что и программа из примера 18.



ВНИМАНИЕ Так как при работе с потоками вывод в файл также производится через буфер, правильно прочитать новое содержимое только что записанного файла можно исключительно после выполнения инструкции `outf1<<flush;` или после выхода из программы.

Имя файла может быть введено с клавиатуры. Для этого в программе должны быть строки:

```
char filename[40];
cout << "\nВведите имя файла:";
cin >> filename;
ofstream outf1 (filename);
if (outf1 == NULL)
    cout << "Ошибка открытия файла" << filename;
```

При возникновении ошибок в процессе работы, например при чтении символа конца файла, значения `inf` и `outf` принимают значения `NULL` (`'\0'`).

Если количество строк в файле не определено, то для работы с ним полезно использовать функцию `getline` объекта `inf`, которая аналогична функциям `gets` и `fgets`. Первым ее параметром является строка, в которую считываются символы из файла, вторым — максимально возможное количество символов в ней.

Пример 20

```
//Копирование файла mytest.txt
//в файл mytest1.txt
#include <fstream.h>
void main()
{
    char wstr[80];
    ifstream myinf ("mytest.txt");
    if (myinf == NULL)
    {
        cout<<"Ошибка открытия файла mytest.txt.";
    }
}
```


Пример 20**(продолжение)**

```
        return;
    }
    ofstream myoutf ("mytest1.txt");
    if (myoutf == NULL)
    while (myinf)
    {
        myinf.getline (wstr, 80);
        myoutf << wstr << '\n';
    }
    else
    {
        cout<<"Ошибка открытия файла mytest1.txt.";
        return;
    }
}
```

Файловые операторы потоков могут быть применены для чтения и записи двоичных файлов, в том числе — для прямого доступа к ним (см. также главу «Объекты и классы библиотеки ввода/вывода»).

Операторы или функции?

В этой главе были описаны два возможных пути для выполнения ввода/вывода (чтение и запись файла также являются операциями ввода/вывода): с помощью функций, объявленных в файле `stdio.h`, и с помощью операторов потока ввода/вывода. Какой из них выбрать? К преимуществам первого способа относится возможность более гибкого управления форматированием и его полная совместимость с C. С другой стороны, используя операторы потоков, можно не задумываться о форматах вывода информации, если это не имеет принципиального значения. Кроме того, этот метод позволяет легко переопределять форму вывода для любого класса объектов.

Самое главное, старайтесь не смешивать эти способы без крайней необходимости. Если без этого обойтись нельзя, не забудьте вызвать функцию `sync_with_stdio()`.

Классы

Для программиста ключевым моментом в С++ является возможность определить свой тип так, чтобы с ним было удобно и просто работать.

С другой стороны, как только новый тип определен, он становится расширением С++, и его можно использовать как любой другой тип, в том числе стандартный. Например, можно определить операцию сложения для нового типа и в дальнейшем использовать для ее выполнения обычный оператор сложения (+). Компилятор сам определяет, что подразумевается под оператором сложения для данных этого типа.

В этом отношении все типы и классы С++ равноправны.

Вопросам определения и создания новых классов посвящена эта глава. В ней также показаны преимущества классов и некоторые особенности их использования.

Создание нового класса

Под *классом* в С++ понимают тип, определенный программистом. Этот термин применим ко всему, что создано с использованием ключевого слова `class`, а также структурам и объединениям, о которых в этой главе будет рассказано позже.

Определение класса выглядит следующим образом:

```
class <имя класса >
{
    private:
        <объявления>
    public:
        <объявления>
    protected:
        <объявления>
};
```




ВНИМАНИЕ В конце объявления класса (как в конце любого объявления переменных) обязательно наличие точки с запятой (;).

Нужно отметить, что объявления могут включать как объявления функций, так и объявления данных, которые соответственно называются *функциями-членами* и *данными-членами*. Данные-члены определяют свойства класса: его длину, адрес, форму; функции-члены определяют его поведение: действия, которые он может выполнять.

Помимо обычных объявлений в определении класса содержатся ключевые слова, задающие область действия (видимости) членов класса:

- `private` (закрытый) — члены класса, объявленные после этого ключевого слова, определены только внутри класса и вне его использоваться не могут;
- `public` (открытый) — члены класса, объявленные после этого ключевого слова, определены как для членов класса, так и вне его (при условии указания имени класса);
- `protected` (защищенный) — члены класса, объявленные после этого ключевого слова, определены для членов класса, его потомков и друзей (см. также раздел «Виды доступа: `public`, `private` и `protected`» в главе «Наследование»).

Ключевые слова `private`, `public` и `protected` иногда называют атрибутами доступа к членам класса.



ВНИМАНИЕ При отсутствии явного указания области видимости для каких-либо членов класса они определяются как закрытые и недоступны вне определения класса.

Примером класса может быть графическая точка на экране монитора. Ее данными-членами являются координаты x , y , а также цвет. В качестве функций-членов можно за-

дать инициализацию точки, определение ее цвета, установку нового цвета.

Пример 21

```
class GraphPoint
{private:
    int x, y, color;
public:
    void init(int wx, int wy, int wcolor);
    int getcolor(void);
    void setnewcolor(int wcolor);
};
```

Определим еще один класс, являющийся расширением строкового типа C++.

Пример 22

```
class MyStr
{
    char String[256];
public:
    char *getpnt(void);
    int length(void);
    void copy(char *s);
    void plus(char *s);
};
```

В качестве данных-членов в этом классе имеется единственная переменная `string`, по умолчанию определенная как `private`, и в качестве функций-членов четыре функции `getpnt`, `length`, `copy` и `plus`, определенные как `public`.

Поведение класса

Определение функций-членов класса практически не отличается от определения обычных функций, за исключением того, что перед ним должно стоять имя класса и два двоеточия (`::`), и выглядит так:

```
<имя класса>::<имя функции>(<список параметров>)
{<тело функции>}
```




ПРИМЕЧАНИЕ Напомним, что в определении функции отсутствуют точки с запятой после ее заголовка и после фигурной скобки, закрывающей тело функции.

Имя класса является неотъемлемой частью определения функции-члена этого класса, так как несколько классов могут иметь функции-члены с одним и тем же именем. Кроме того, именно функция класса определяет поведение класса, то есть его действия.

Определим функции класса `MyStr`, введенного в предыдущем разделе.

Пример 23

```
#include <string.h>
#include <mystr.h>
char *MyStr::getptr(void)
//Функция определяет адрес строки данных
{
    return String;
}
int MyStr::length(void)
//Функция определяет длину строки данных
{
    return strlen(String);
}
void MyStr::copy(char *s)
/*Функция копирует строку-параметр в область данных
своего класса*/
{
    strcpy(String, s);
}
void MyStr::plus(char *s)
/*Функция дописывает строку-параметр к строке,
находящейся в области данных ее класса*/
{
    strcat(String, s);
}
```


Таким образом, объявив класс `MyStr` в примере 22 и определив все функции, входящие в него, в примере 23, мы ввели новый класс `MyStr`.

Если новый класс нужен только в конкретной программе и не рассчитан на широкое применение, его объявление и определение его функций могут быть записаны в одном файле, например `mustr.cpp`. Этот файл должен содержать полные тексты примеров 22 и 23.



ПРИМЕЧАНИЕ Следует учесть, что в этом файле пока находится только объявление нового класса и определение его функций. Для использования этого класса нужно еще немного дописать текст файла `mustr.cpp` (см. раздел «Объекты класса»).

Если предполагается использование нового класса в других программах, лучше оформить его так, как это обычно принято в C++. Для этого определение класса `MyStr` (весь текст примера 22) сохраняется в виде файла заголовков, например `mustr.h`, а описание функций-членов класса (весь текст примера 23) — в другом файле, например `mustr.cpp`. При использовании объектов класса `MyStr` файл заголовков включается в программу с помощью директивы `#include <mustr.h>`, а файл `mustr.cpp` компилируется и в виде объектного файла `mustr.obj` включается в проект программы (см. также главу «Директивы, макросы и операторы препроцессора»). Далее он используется так же, как все остальные классы C++.

Объекты класса

В предыдущих разделах был определен новый класс и его поведение. Далее имя класса можно использовать как имя любого другого типа C++ (`int`, `float`, `char`), объявив, например, `MyStr str1, str2, wstr;`. Все объявленные переменные являются объектами класса `MyStr`. Но от обычной переменной объект отличается тем, что он может поддерживать вызов функций-членов своего класса и другие операции.

Вызов функции-члена класса имеет следующий синтаксис:
<объект>. <имя функции-члена > (<параметры>);

Пример 24

```
#include <stdio.h>
#include <mystr.h>
void main()
{
    MyStr str1, str2;
    str1.copy(«Привет, »);
    str2.copy(«объект!»);
    str1.plus (str2.getptr());
    puts (str1.getptr());
}
```

В данном примере первые две инструкции после инструкции объявления копируют строки-параметры в поля данных объектов `str1` и `str2` соответственно. Третья дописывает к строке объекта `str1` строку объекта `str2`. Четвертая получает указатель на строку объекта `str1` и выводит ее на экран. Каждый объект имеет свое собственное поле данных.

Таким образом, в результате работы программы на экране монитора появится строка:

Привет, объект!

Если объекты различных классов имеют функции с одинаковыми именами, компилятор определяет нужную функцию через класс объекта. В примере 24 компилятор определяет типы `str1` и `str2` (это `MyStr`) и вызывает функции `MyStr::copy`, `MyStr::plus`, `MyStr::getpnt`. Таким образом, первые две выполняемые инструкции программы обращаются к одной и той же функции класса.



ВНИМАНИЕ Работа с функциями и данными объектов в C++ реализуется по-разному: все объекты одного класса имеют общий доступ к одним и тем же функциям, но каждый из них имеет свое поле данных, независимо от того, описано оно как `public` или `private`.



ПРИМЕЧАНИЕ Класс может содержать несколько данных-членов, но и тогда каждый объект этого класса после объявления будет иметь свою область памяти для размещения этих данных.

Указатели на объекты

Как и для любой переменной в C++ можно определить указатель на объект (адрес объекта в памяти). Это особенно важно при создании объекта и освобождении памяти от него во время работы программы. Для динамического создания объектов в C++ используются операторы `new` и `delete` (см. также раздел «Использование операторов `new` и `delete`» в главе «Указатели, массивы и строки»).

Пример 25

```
/*Фрагмент программы с динамическим созданием и  
уничтожением объекта класса MyStr*/  
MyStr *pstr1; //объявление указателя на объект  
pstr1=new MyStr; //выделение памяти для объекта  
... //работа с объектом  
delete pstr1; //освобождение памяти
```

Для выделения памяти под объекты можно использовать функции `malloc` и `free` (см. раздел «Использование стандартных функций `malloc` и `free`» в главе «Указатели, массивы и строки»), но использование операторов `new` и `delete` предпочтительнее, так как последние автоматически вызывают конструкторы и деструкторы (о них в этой главе будет рассказано позднее).

Для работы с членами объектов, созданных динамически, используется специальный оператор (`->`). При работе с объектом, созданным в примере 25, допустимы следующие инструкции:

```
pstr1->copy(«Привет»);  
pstr1->plus(«, объект!»);  
puts (pstr1->getptr());
```


Если бы член `String` класса `MyStr` был объявлен как `public`, к нему можно было бы обратиться так:

```
pstr1->String
```

Достоинства классов

Описав объект `str1` класса `MyStr` и скопировав в него строку с помощью инструкции `str1.copy(«Привет»);`, резонно задать вопрос, в чем преимущество этого метода по сравнению с использованием обычной функции `strcpy(str1, «Привет»)` для переменной `str1`, определенной как `char str1[256];`?

Можно выделить, по крайней мере, одно преимущество первого метода. При обычном подходе переписывание какого-либо модуля, входящего в проект, всегда является делом рискованным и чревато возникновением ошибок в программах, использующих этот модуль. При работе с классами без каких-либо опасений можно заменять закрытые данные-члены класса и переписывать открытые функции-члены класса при сохранении списка их параметров и типа возвращаемого значения.

Таким образом, объектно-ориентированное программирование обеспечивает простоту и безопасность исправления, обновления и замены фрагментов программ, что особенно важно для больших проектов.

Закрытые данные-члены класса и определения всех функций-членов класса входят во внутреннюю, исполнительную часть класса (*implementation*). В то же время для взаимодействия с внешним миром часть членов класса должна быть объявлена открытой (*public*). Эта часть класса называется ее *интерфейсом* (*interface*).



ПРИМЕЧАНИЕ В отличие от прототипов функций-членов класса, объявленных открытыми, определения этих функций не входят в интерфейс класса.

Возможность изменения исполнительной части класса без опасности возникновения ошибок в программах, использующих этот класс, называется *инкапсуляцией*.

Чтобы изменение класса не требовало изменения программ, использующих этот класс, интерфейсная часть класса не должна изменяться.

Класс `MyStr`, введенный в примере 22, обладает одним недостатком: строка, определенная в нем, занимает 256 байт. Это неэффективно для коротких строк и недостаточно для длинных. В последнем случае последствия могут быть катастрофическими. Заменим собственные данные класса `MyStr` на указатель строки и введем ее длину `lstr` (см. также раздел «Использование операторов `new` и `delete`» в главе «Указатели, массивы и строки»).

Пример 26

```
//-----  
//Запишем в файл mystr.h определение класса  
class MyStr  
{  
    char *pstr;  
    int lstr;  
public:  
    char *getptr(void);  
    int length(void);  
    void copy(char *s);  
    void plus(char *s);  
};  
//-----  
//Запишем в файл mystr.cpp определения функций  
#include <string.h>  
char *MyStr::getptr(void)  
//Функция определяет адрес строки данных  
{  
    return (pstr);  
}  
int MyStr::length(void)  
//Функция определяет длину строки данных  
{  
    return (lstr);  
}
```


Пример 26

(продолжение)

```
void MyStr::copy(char *s)
/*Функция копирует строку-параметр в область данных
своего класса*/
{
    int n;
    n = strlen(s);
    if (lstr != n)
    {
        if (pstr) //если pstr - не пустая строка
            delete [] pstr;
        pstr=new char[n+1]; //длина строки+1
        lstr=n;
    }
    strcpy(pstr, s);
}

void MyStr::plus(char *s)
/*Функция дописывает строку-параметр к строке,
находящейся в области данных ее класса*/
{
    int n;
    char *ptmp;
    n = strlen (s);
    if (n==0) return;
    ptmp = new char[lstr+n+1]; //длина строки+1
    if (pstr) //если pstr - не пустая строка
    {
        strcpy (ptmp, pstr);
        delete [] pstr;
    }
    strcat(ptmp, s);
    pstr=ptmp;
    lstr+=n;
}
```

Несмотря на то что закрытые данные класса и открытые функции значительно изменились, все программы, использующие этот класс, не потребуют никаких изменений. Про-

грамму из примера 24 также не потребуются менять, однако, во-первых, для коротких строк она будет работать более эффективно, во-вторых, она вполне подойдет и для строк длиннее 256 байт.



ВНИМАНИЕ Динамическое распределение памяти требует очень внимательного отношения. Каждому оператору `new` должен соответствовать оператор `delete`, иначе неизбежны потери памяти в системе, что в конечном итоге может привести к нарушениям в работе системы и потере или порче данных.

Конструкторы и деструкторы: рождение и смерть объектов

Класс `MyStr`, несмотря на возросшую эффективность, имеет два недостатка. Один из них обуславливается тем, что из-за неопределенности значения указателя `pstr` функции класса не будут работать корректно, пока в объект не скопирована какая-нибудь строка. Другой заключается в том, что память, выделенная объектам при последнем обращении к ним, освобождается только при выходе из программы.

Таким образом, необходимо выполнить дополнительные действия при создании объекта и при его удалении. К счастью, в C++ предусматривается возможность инициализации и удаления объектов с помощью, соответственно, *конструкторов* и *деструкторов*. Это специальные функции-члены класса, которые управляют процессом создания и удаления объектов и имеют необычный синтаксис:

- имя конструктора совпадает с именем класса, и выглядит он следующим образом: `<имя класса>;`
- имя деструктора тоже совпадает с именем класса, и выглядит он следующим образом: `~<имя класса>.`

Объявим для класса `MyStr` конструктор `MyStr` и деструктор `~MyStr`. Тогда объявление класса будет выглядеть так:


```
class MyStr
{
    char *pstr;
    int lstr;
public:
    MyStr(); //конструктор
    ~MyStr(); //деструктор
    char *getptr(void);
    int length(void);
    void copy(char *s);
    void plus(char *s);
};
```

Как и в примере 26 это объявление можно записать в файл `mystr.h`.



ВНИМАНИЕ Конструкторы и деструкторы отличаются от обычных функций: они не возвращают никакого типа (даже `void`) и при отсутствии параметров имеют пустой список `()` вместо `(void)`.

В файл `mystr.cpp` добавим определения конструктора и деструктора класса `MyStr`.

```
MyStr::MyStr()
{
    pstr = new char[1];
    *pstr = '\0'; //или strcpy(pstr, «»);
    lstr = 0;
}
MyStr::~MyStr()
{
    delete [] pstr;
}
```

В определении конструктора отводится память для пустой строки (1 байт для нуль-терминатора), определяется ее содержимое (пустая строка) и длина (0).

Деструктор освобождает последний выделенный блок памяти.

В отличие от деструкторов, действия которых сводятся обычно к освобождению памяти, конструкторы могут выполнять значительно более разнообразные действия (см. также главу «Конструкторы»).

Определение встраиваемой функции

Инкапсуляция иногда может привести к неэффективности программы, так как даже если функция состоит только из одной инструкции, ее вызов происходит стандартным образом — через стек. Примером таких функций являются функции `MyStr::getpnt` и `MyStr::length`.

В C++ для таких случаев предусмотрено весьма эффективное средство — *встраиваемые* (`inline`) функции. Встретив вызов такой функции, компилятор подставляет (встраивает) ее тело вместо вызова, минуя стандартное обращение к функции через команду процессора `CALL`. Для очень коротких функций такой подход является идеальным, так как сохраняет инкапсуляцию и делает программу эффективнее.

Пример 27

```
class MyStr
{
    char *pstr;
    int lstr;
public:
    MyStr(); //конструктор
    ~MyStr(); //деструктор
    char *getpnt(void){return pstr;};
    int length(void){return lstr;};
    void copy(char *s);
    void plus(char *s);
};
```

Несмотря на краткость, функции `getpnt` и `length` являются обычными функциями и подчиняются тем же синтаксическим правилам, что и остальные функции. Каждая из

них может содержать больше одной инструкции, но чаще всего эти функции очень короткие.



ПРИМЕЧАНИЕ Как и в обычных определениях функций, после таких определений точка с запятой не ставится (см. также раздел «Функции» в главе «Основные особенности программирования на C++»).

C++ поддерживает ключевое слово `inline`, предназначенное для того, чтобы встраивать тело любой функции непосредственно в код программы, не выполняя стандартного обращения. Однако в объявлении класса такое встраивание делается автоматически.

Структура как разновидность класса

Особенностью C++ является то, что в структурах (типах, определенных через ключевое слово `struct`) и объединениях (типах, определенных через ключевое слово `union`) допустимо использование функций, аналогичное их использованию в определении классов. Эта возможность делает структуры и объединения очень похожими на классы. Однако у них имеется одно отличие:

- все члены класса являются по умолчанию закрытыми (`private`);
- все члены структуры являются по умолчанию открытыми (`public`), но их можно переопределить как `private` или `protected`;
- все члены объединения являются по умолчанию открытыми (`public`), и явное переопределение атрибута доступа к ним невозможно.

Таким образом, структуры и объединения являются особыми разновидностями классов и сохранены в C++ только для совместимости с C. Для поддержания инкапсуляции в C++ рекомендуется определение новых типов с помощью ключевого слова `class`.

Конструкторы

В предыдущей главе было введено понятие *конструктора класса*. Конструктор — это функция-член класса, которая создает объекты данного класса, выделяя для них память и/или инициализируя их.

Однако для корректного определения класса часто необходимо иметь несколько конструкторов. Все они имеют одно и то же имя, совпадающее с именем класса, но отличаются числом и типом параметров.

Одним из наиболее важных конструкторов является *конструктор копирования*. Он используется для копирования объектов и передачи их в качестве возвращаемого значения функции.

Обо всех особенностях определения конструкторов рассказывается в этой главе.

Перегрузка конструкторов

После определения класса `MyStr` требуется инициализировать объекты этого класса. Это можно сделать разными способами: просто задавая имя объекта, без инициализации, определяя строку в качестве начального значения поля `pstr`, или определяя один объект равным другому.

Каждое из этих объявлений требует своего варианта конструктора. C++ позволяет написать столько конструкторов, сколько необходимо, и прототипы конструкторов для инициализации объектов вышеперечисленными способами выглядят так:

```
MyStr (); //без инициализации
```

```
MyStr (char *wstr); //инициализация строки
```

```
MyStr (MyStr &wstr); /*инициализация из другого объекта*/
```

В зависимости от контекста (типа и числа параметров конструктора) компилятор вызывает нужный вариант конст-

руктора. Эта особенность называется *перегрузкой конструкторов*.



ПРИМЕЧАНИЕ Перегрузка (overloading) широко используется в C++ не только для конструкторов и функций-членов класса. Практически любые функции C++ могут быть перегружены. Перегрузка означает, что одним и тем же именем могут обозначаться разные варианты функций. Выбор нужного варианта выполняется в процессе компиляции в зависимости от числа и типа параметров.

Рассмотрим первые два конструктора более подробно, третьему будет посвящен отдельный раздел в этой главе.

Первый конструктор введен в разделе «Конструкторы и деструкторы: рождение и смерть объектов» главы «Классы»; он отводит один байт для строки, инициализирует пустую строку и определяет длину пустой строки равной нулю:

```
MyStr::MyStr()
{
    pstr = new char[1];
    *pstr = '\0';
    lstr = 0;
}
```

Второй конструктор создает объект класса MyStr из обычной строки языка C. Основные требования к этому конструктору сохраняются: он должен выделять память для строки, инициализировать эту строку и записывать ее длину в поле lstr.

```
MyStr::MyStr(char *wstr)
{
    pstr = new char[strlen(wstr)+1];
    strcpy(pstr, wstr);
    lstr = strlen(wstr);
}
```

Определив эти конструкторы, можно инициализировать объекты класса MyStr следующим образом:


```
MyStr str1; //без инициализации  
MyStr str2("Привет!"); /*инициализация строки*/
```

Конструктор по умолчанию

Прежде чем перейти к третьему конструктору, рассмотрим специальный вид конструктора — *конструктор по умолчанию*. Конструктор по умолчанию — это конструктор без параметров. Каждый класс должен иметь такой конструктор.

 **ВНИМАНИЕ** Если для класса явно не определено никаких конструкторов, компилятор поддерживает *скрытый конструктор* — конструктор по умолчанию, который инициализирует данные-члены нулями. Но если для класса определен хотя бы один конструктор, конструктор по умолчанию исчезает.

Таким образом, в определение класса всегда следует включать конструктор без параметров, даже если он не выполняет никаких операций. После его явного определения он подставляется вместо конструктора по умолчанию и уже не может исчезнуть, даже если будет определен еще какой-либо конструктор.

Следует отметить, что конструктор по умолчанию используется во многих ситуациях: при уже упомянутом определении переменных без инициализации, объявлении массива без инициализации элементов, при задании оператора new без параметров. Если конструктор по умолчанию отсутствует, то во всех этих ситуациях возникает ошибка.

 **ПРИМЕЧАНИЕ** Конструкторы по умолчанию были введены в C++ для совместимости с простейшими классами C (в частности, struct), которые не могли иметь функций-членов.

Конструктор копирования

Конструктор копирования MyStr (MyStr&) — еще один вид конструктора, без которого трудно обойтись. Он нужен для инициализации объекта с помощью другого объекта того же класса. Например:


```
MyStr str2("Привет!"); /*вызов MyStr(char*)*/  
MyStr str3(str2); /*инициализация str3, копии str2*/
```

Конструктором копирования называется конструктор, который используется для создания копии другого объекта того же класса. Этот конструктор также очень важен, так как помимо объявления переменных он требуется в следующих ситуациях:

- когда объект класса как параметр функции передается по значению (C++ вызывает конструктор копирования для создания копии аргумента, которая потом размещается в стеке):

```
void disp_str(MyStr wstr);
```

- когда объект класса является возвращаемым значением функции (C++ вызывает конструктор копирования для создания объекта, который возвращается вызвавшей функцией):

```
MyStr func(int wn);
```

Так как эти ситуации возникают достаточно часто, компилятор поддерживает простейший скрытый конструктор копирования. Он просто копирует данные-члены одного объекта в данные-члены другого.

В некоторых случаях этого вполне достаточно, но не всегда. Например, для класса `MyStr` использование такого конструктора копирования может привести к печальным последствиям.

Допустим, для данных-членов объекта `str3` выполнено копирование данных-членов объекта `str2`. Тогда указатели `str3.pstr` и `str2.pstr` указывают на один и тот же блок памяти. Таким образом, изменение содержимого строки одного объекта ведет к изменению строки другого объекта. Но наиболее опасная ситуация возникает, когда один из объектов удаляется, и вместе с ним удаляется строка, адрес которой находился в поле `pstr` этого объекта. В этом случае поле `pstr` другого объекта является указателем на неопределенную область, что приводит к ошибке выполнения программы.

Пример 28

```
pstr1 = new MyStr;  
pstr1->copy ("Привет!");  
MyStr str2 (pstr1);  
//...  
delete pstr1; //удаление строки данных  
puts (str2.getptr()); //Ошибка!
```

Следует отметить, что при удалении объекта информация в памяти, отведенной для него, изменяется не сразу, а только при следующем динамическом выделении памяти.

Ошибку при выполнении последней строки примера 28 может вызвать даже простое копирование в объект с указателем `pstr1` нового значения строки с помощью функции `copy`:

```
pstr1->copy ("Новая строка");
```

В этом случае также происходит освобождение памяти и ее новое выделение (см. также пример 27 в главе «Классы»).

Таким образом, основная цель копирования должна состоять в том, чтобы при потере или удалении объекта-оригинала его копия продолжала полноценное существование и функционирование. Следовательно, конструктор копирования должен выполнять те же действия, что и конструктор `MyStr(char*)`: выделять новый блок памяти и лишь затем выполнять копирование строки данных и формирование длины строки.

Ссылки и конструктор копирования

Сначала необходимо разобраться в том, что означает оператор ссылки (`&`) и как он используется при копировании.



ПРИМЕЧАНИЕ Объявление ссылки и определение адреса — два разных действия, несмотря на то, что для их обозначения используется один и тот же символ амперсанда (`&`).

Амперсанд используется для выполнения логической и побитовой операции И. Применение этого символа в раз-

личных операторах является примером перегрузки последних (см. также главу «Перегрузка операторов»).

При объявлении переменной наличие амперсанда означает, что эта переменная используется в качестве ссылки (дополнительного имени) другой переменной.

Пример 29

```
int x, &y = x; //y – ссылка на x
x = 0;
y = 10; //означает, что x тоже равен 10
```

Этот фрагмент эквивалентен следующему:

```
int x, *y;
y=&x; //y – ссылка на x
x = 0;
*y = 10; //означает, что x тоже равен 10
```

Внимательно сравнивая эти два фрагмента, можно отметить, что при определении переменной *y* в виде ссылки (&*y*) в первом варианте, она выглядит как обычная целая переменная, и то, что фактически она является указателем на переменную *x*, скрыто синтаксисом C++.

Можно указывать амперсанд просто для создания дополнительного имени переменной, но наиболее важно его использование при вызове функций. Рассмотрим прототип конструктора копирования класса *MyStr*:

```
MyStr (MyStr &wstr);
```

Этот конструктор имеет один параметр *wstr*, типом которого является ссылка на объект класса *MyStr*. Проще говоря, чтобы создать копию объекта, конструктор копирования получает ссылку на объект и через нее — доступ ко всем членам этого объекта.



ВНИМАНИЕ Разница между передачей параметра как указателя и передачей его по ссылке заключается в последовательности действий. При передаче указателя данные уже определены и для них определяется адрес (указатель). При передаче по ссылке сначала определяется адрес, а затем по этому адресу формируются данные.

С этой точки зрения все, что нужно помнить об операторе ссылки — это то, что параметр берется как значение, а не указатель, несмотря на то что он передается по ссылке. Конструктор копирования в этом случае будет выглядеть так:

```
MyStr::MyStr(MyStr &wstr)
{
    char *tmpstr;
    int l;
    tmpstr = wstr.getptr();
    l = wstr.length();
    pstr = new char[l+1];
    strcpy (pstr, tmpstr);
    lstr=l;
}
```

Этот конструктор во многом подобен конструктору `MyStr(char*)`. Принципиальное отличие заключается в том, что конструктор копирования должен сначала выделить строку и ее длину во вспомогательных переменных, а затем уже с их помощью произвести копирование.

Заголовок конструктора может на первый взгляд показаться странным из-за троекратного вхождения в него идентификатора `MyStr`, но каждое употребление `MyStr` в заголовке конструктора копирования играет свою роль: первое указывает на класс, для которого определяется конструктор, второе — на имя конструктора, третье — на тип аргумента.

Защита параметра от изменений

Наконец, осталось уточнить одну небольшую деталь. Обычно передача параметра по ссылке употребляется в том случае, если нужно его изменять. Однако в нашем случае изменение объекта, копия которого создается с помощью конструктора копирования, крайне нежелательно. Для таких случаев (передача параметра по ссылке и одновременный запрет его изменения) C++ предлагает использование ключевого слова `const` при объявлении параметров.

Итак, окончательный вариант конструктора копирования выглядит так:

```
MyStr::MyStr(const MyStr &wstr)
{
    //...
    //тело конструктора сохраняется
}
```

Теперь конструктор копирования не может изменить значения ни одного из членов объекта `wstr`, так как любая попытка присвоить значение объекту `wstr` или любому из его данных-членов вызовет ошибку на этапе компиляции. Но существует еще одно слабое место у этого конструктора: вызов функций `getptr` и `length` для объекта `wstr`. Как обеспечить неизменность параметра или его членов при вызове этих функций? Решением проблемы является объявление функций `getptr` и `length` *функциями-константами*.

```
char *MyStr::getptr(void)
const { return (pstr);}
int MyStr::length(void)
const { return (lstr);}
```

Объявление функций константами делает безопасным их вызов для объектов, объявленных константами, и предохраняет последние от каких-либо изменений в функциях, использующих эти объекты в качестве параметров.

Примеры конструкторов

Создадим несколько конструкторов для класса графической точки, введенного в примере 21 предыдущей главы.

Пример 30

```
class GraphPoint
{private:
    int x, y, color;
public:
    GraphPoint (); //конструктор по умолчанию
    GraphPoint (int wx, int wy, int wcolor);
```



```
    GraphPoint (const GraphPoint &wgp);  
    /*конструктор копирования*/  
    int getcolor(void);  
    void setnewcolor(int wcolor);  
};
```

Класс GraphPoint значительно проще класса MyStr. Он не требует динамического распределения и перераспределения памяти, а также ее освобождения, поэтому конструктор по умолчанию может быть пустым. Он включается в определение класса только для того, чтобы избежать ошибок при неявном вызове конструктора этого типа. Таким образом, последний имеет вид:

```
GraphPoint::GraphPoint () {}
```

Следующий конструктор более интересен, он инициализирует на экране монитора графическую точку с координатами wx, wy цвета wcolor.

```
GraphPoint::GraphPoint(int wx,int wy,int wcolor)  
{  
    x = wx;  
    y = wy;  
    color = wcolor;  
}
```

Наконец, конструктор копирования инициализирует объект, копируя в него данные-члены другого объекта этого класса. Объект, копия которого создается, передается конструктору в качестве параметра. Этот конструктор может отсутствовать в определении класса, так как он выполняет те же действия (почленное копирование), что и конструктор копирования, поддерживаемый компилятором.

```
GraphPoint::GraphPoint(const GraphPoint &wgp)  
{  
    x = wgp.x;  
    y = wgp.y;  
    color = wgp.color;  
}
```


Далее приводятся примеры вызова каждого из описанных выше конструкторов:

```
GraphPoint gp1;  
GraphPoint gp2(100, 50, YELLOW);  
GraphPoint gp3 (gp2);
```

Вызов конструкторов и конверсия

Итак, мы разобрали механизм вызова конструкторов во время работы программы. Что происходит при объявлении объектов с их одновременной инициализацией? Рассмотрим следующие объявления переменных.

Пример 31

```
MyStr str1 ("Привет!");  
MyStr str2 = str1;  
MyStr str3 = "Привет!";
```

Понятно, что в первом объявлении вызывается конструктор `MyStr (char *)`. Любопытно, что во втором объявлении также вызывается конструктор (конструктор копирования), и эта строка эквивалентна показанной ниже строке не только по результату работы, но и по способу ее выполнения:

```
MyStr str2 (str1);
```

При выполнении обеих строк вызывается один и тот же конструктор копирования `MyStr (MyStr &)`.

Следует отметить, что в C++ присваивание и инициализация выполняются совершенно по-разному, несмотря на внешнее сходство операторов (=).



ВНИМАНИЕ За исключением объявления переменных знак равенства всегда обозначает присваивание. При объявлении переменных он указывает на инициализацию, которая вызывает соответствующий типу переменной конструктор, а не оператор присваивания.

В главе «Перегрузка операторов» будет подробно рассказано о том, как программист может определить поведение

оператора присваивания, но это совсем не то же самое, что конструктор копирования. Иногда (но далеко не всегда) они действительно выполняют одни и те же действия. (Главным отличием является то, что конструктор копирования должен, как правило, сначала создать объект, а уж затем присваивать его членам начальные значения.)

Аналогичным образом в последней строке примера 31 вызывается конструктор `MyStr (char*)`, и она эквивалентна первой строке этого же примера.

Существует еще один случай, когда вызываются конструкторы, — при преобразовании параметра какого-либо типа в класс. Однако данная процедура выполняется только в том случае, если функция имеет ровно один параметр и существует конструктор класса из переменной этого типа.

Пример 32

```
/*Прототип функции, удаляющей пробелы из строки,  
являющейся членом класса MyStr*/  
MyStr Delete_Spaces (MyStr wstr);  
//фрагмент вызывающей функции  
MyStr str1;  
str1 = Delete_Spaces ("123");
```

Последняя строка примера выполняет вызов конструктора `MyStr (char*)` для создания объекта класса `MyStr` из строки "123". Результирующий объект класса `MyStr` передается функции `Delete_Spaces`, и далее в нем удаляются пробелы.

Перегрузка операторов

Одной из наиболее интересных возможностей C++ является *перегрузка операторов* (operator overloading), что означает возможность переопределения практически любого оператора C++. Такая особенность позволяет с помощью определения новых типов и операций над ними приблизить синтаксис C++ к синтаксису других, возможно более привычных для программиста, языков.

Возможности по определению новых операторов практически не ограничены и зависят только от комбинаций возможных типов переменных, участвующих в вычислениях.

Базовый синтаксис

Существует множество вариантов *функций операторов*, но их базовый синтаксис очень прост. Для данного оператора @ имя его функции определяется так:

operator@

Таким образом, функции операторов для операторов +, -, *, / соответственно имеют имена:

operator+

operator-

operator*

operator/

Эти определения выглядят очень просто, но дальнейшие действия зависят от следующих факторов:

- имеет оператор один или два операнда (некоторые операторы, например минус, могут быть использованы в обоих вариантах: как с двумя, так и с одним операндом);
- если оператор имеет два операнда, то может ли каждый из них находиться как перед, так и после оператора, или его позиция фиксирована, и он располагается

только перед оператором или только после него. При возможности использования операндов по обе стороны оператора понадобятся *функции-друзья*, обеспечивающие доступ к классу с помощью ключевого слова `friend`.

Функции-друзья обеспечивают доступ к членам класса без использования его функций-членов (см. также раздел «Для чего нужны функции-друзья»).

Синтаксис для функций сложения может быть описан следующим образом.

Пример 33

```
class MyStr
{
    //...
    friend MyStr operator+(MyStr wstr1, MyStr wstr2);
    friend MyStr operator+(MyStr wstr1, char *wstr2);
    friend MyStr operator+(char *wstr1, MyStr wstr2);
};
MyStr operator+(MyStr wstr1, MyStr wstr2)
{
    //...
}
MyStr operator+(MyStr wstr1, char *wstr2)
{
    //...
}
MyStr operator+(char *wstr1, MyStr wstr2)
{
    //...
}
```



ПРИМЕЧАНИЕ Вне объявления класса ключевое слово `friend` использовать не нужно.

В этом примере видна тесная связь между перегрузкой функций и перегрузкой операторов. Как и при перегрузке обычных функций вариант функции оператора выбирается в зависимости от количества и типа параметров.

Создание функций оператора сложения

Определим функцию оператора сложения для двух параметров, имеющих тип `MyStr` как конкатенацию двух строк.

Пример 34

```
MyStr operator+(MyStr wstr1, MyStr wstr2)
{
    MyStr tmpstr(wstr1);
    tmpstr.plus(wstr2.getptr());
    return tmpstr;
}
```

Результат выполнения оператора сложения двух значений типа `MyStr` также имеет тип `MyStr`.

В первой строке тела функции оператора создается копия объекта `wstr1` с помощью конструктора копирования, затем к строке этого объекта дописывается строка объекта `wstr2`.

Наконец, в последней строке полученный результат возвращается в вызывающую функцию. Инструкция `return` создает копию объекта `tmpstr`, неявно вызывая конструктор копирования, передает ее в стек и удаляет оригинал. Затем вызывающая функция получает результат из стека (см. также раздел «Конструктор копирования» в главе «Конструкторы»).

Конструктор копирования для класса `MyStr` является довольно сложным, но необходимым, так как конструктор копирования, используемый компилятором по умолчанию и выполняющий прямое почленное копирование объекта, приведет к возникновению ошибок при выполнении программы, а возможно, и к «зависанию» системы.

Когда компилятор C++ встречает в программе (функции) выражение `str1+str2`, он транслирует его в вызов функции соответствующего оператора. Если такой функции нет, то во время компиляции выдается сообщение о том, что операция не определена.

Если функция найдена, она получает переменные `str1` и `str2` в качестве параметров, выполняет описанные выше действия, возвращает результат, который с помощью оператора сложения передается объекту программы (функции), вызвавшей эту функцию.

Оставшиеся две функции оператора сложения выглядят так.

Пример 34**(продолжение)**

```
MyStr operator+(MyStr wstr1, char *wstr2)
{
    MyStr tmpstr(wstr1);
    tmpstr.plus(wstr2);
    return tmpstr;
}
MyStr operator+(char *wstr1, MyStr wstr2)
{
    MyStr tmpstr(wstr1);
    tmpstr.plus(wstr2.getptr());
    return tmpstr;
}
```

Все эти функции похожи. В каждой из них при вычислении суммы строк происходит следующее:

- создается новый объект типа `MyStr`, который инициализируется как копия первого параметра;
- к нему дописывается содержимое второго параметра;
- возвращается результирующий объект класса `MyStr`.

Для чего нужны функции-друзья

Унарным оператором называется оператор с одним операндом, бинарным — оператор с двумя операндами. При написании функций для бинарных операторов могут быть полезными функции-друзья.

Обычно глобальные функции не имеют доступа к закрытым членам классов. Для обхода этого ограничения используются функции-друзья, определяемые с помощью ключевого слова `friend` и обеспечивающие доступ ко всем членам класса вне области его определения.

В примере 30 был введен класс `GraphPoint`, данные-члены которого `x`, `y` и `color` являются закрытыми членами этого класса. Для создания объектов класса `GraphPoint` был описан конструктор:

```
GraphPoint::GraphPoint(int wx, int wy, int wcolor);
```


Однако определение данных-членов объектов этого класса можно выполнить и с помощью функции-друга `newpoint` класса `GraphPoint`.

Пример 35

```
class GraphPoint
{
    private:
        int x, y, color;
    friend void newpoint(GraphPoint gp, int wx, int wy, int
        wcolor);
    public:
        GraphPoint (); //конструктор по умолчанию
    //...
    //все конструкторы и функции из примера 30
    главы «Конструкторы»
};
friend void newpoint(GraphPoint gp, int wx, int wy,
    int wcolor)
{
    gp.x = wx;
    gp.y = wy;
    gp.color = wcolor;
}
```

Объявление функции `newpoint` как функции-друга делает возможным использование в ней внутренних переменных `x`, `y` и `color` класса `GraphPoint`. Ключевое слово `friend` позволяет написать такие функции, как `operator+(char*, MyStr)`. Можно реализовать функции оператора как функции-члены, без использования ключевого слова `friend`, например, функция оператора сложения `operator+ (MyStr, char*)` может быть определена как функция-член класса `MyStr`. Однако такие функции предполагают, что объект класса может находиться только перед оператором.

Пример 36

```
class MyStr
{
    //...
```



```
    MyStr operator+(char *wstr);  
};  
MyStr MyStr::operator+(char *wstr)  
{  
    MyStr tmpstr(*this);  
    tmpstr.plus(wstr);  
    return tmpstr;  
}
```

При использовании этого способа сложение транслируется в вызов функции-члена класса, определяемого по левому операнду, то есть в нашем случае через функцию-член класса `MyStr`.

Встретив выражение `str+s`, где `str` является объектом класса `MyStr`, а `s` имеет тип `char*`, компилятор транслирует его в вызов функции `str.operator+(s)`, описанной как `MyStr MyStr::operator+(char *wstr)`.

Первая инструкция тела функции инициализирует новый объект класса `MyStr` через объект `*this` с помощью конструктора копирования.

В C++ ключевое слово `this` является указателем на текущий объект, а `*this` — ссылкой на этот объект. Таким образом, первая инструкция тела функции оператора сложения инициализирует объект `tmpstr`, копируя содержимое левого операнда с помощью ссылки `*this` (см. также раздел «Указатель `this` и его использование при создании функций»). В этом случае положение объекта класса `MyStr` ограничивается возможностью расположения только перед операндом. Таким образом, функцию для вычисления выражения `s+str`, где `s` имеет тип `char*`, а `str` является объектом класса `MyStr`, можно реализовать только с помощью функций, не являющихся членами класса.

По этой причине бинарные операторы обычно реализуются как функции-друзья, а не как функции-члены. Таким образом, мы возвращаемся к необходимости объявления класса и функций оператора сложения, приведенного в примере 33:


```
class MyStr
{
    //...
    friend MyStr operator+(MyStr wstr1, MyStr wstr2);
    friend MyStr operator+(MyStr wstr1, char *wstr2);
    friend MyStr operator+(char *wstr1, MyStr wstr2);
};
```

Создание функции оператора присваивания

Присваивание (=) является специальной операцией, для которой компилятор поддерживает скрытую функцию по умолчанию, если она не определена в явном виде. По умолчанию присваивание объектов одного типа выполняется путем копирования данных-членов одного объекта в данные-члены другого.



ПРИМЕЧАНИЕ Можно избежать присваивания всех значений данных-членов, заново определив функцию оператора присваивания и сделав ее закрытой для класса.

Следующий фрагмент программы должен быть верен, поскольку, как правило, C++ поддерживает выполнение присваивания объектов одного типа, так же как в C поддерживается прямое присваивание структур:

```
MyStr wstr1, wstr2;
//...
wstr1 = wstr2;
```

Однако, как и в случае с копированием, присваивание, поддерживаемое компилятором по умолчанию, выполняется неверно для классов, использующих динамическое распределение памяти. Примером такого класса может служить класс `MyStr`, для которого простое почленное копирование может привести к ошибкам во время выполнения программы (см. также раздел «Конструктор копирования» в главе «Конструкторы»).

Следовательно, при создании класса, подобного классу `MyStr`, необходимо тщательно задать поведение оператора присваивания. Функция этого оператора должна быть чле-

ном класса со следующим прототипом, входящим в объявление класса:

```
<имя класса> operator=(const <имя класса> &<param>);
```

Имя переменной `param` в этом определении является произвольным. Соответствующее определение функции имеет синтаксис:

```
<имя класса>& <имя класса>::operator=(const <имя класса>
    &<param>)
{<тело функции>}
```

Для класса `MyStr` прототип функции оператора присваивания может иметь следующий вид:

```
class MyStr
{
    ...
    MyStr& operator=(const MyStr &source);
};
```

В этом определении `operator=` — имя функции, `MyStr&` — тип возвращаемого ею значения, ссылка на объект класса `MyStr`, `const MyStr &` — тип параметра `source`, ссылка на объект класса `MyStr`. Так как параметр описан с помощью ключевого слова `const`, функция не может изменить его.

Наиболее короткая версия определения функции оператора присваивания выглядит так:

```
MyStr& MyStr::operator=(const MyStr &source)
{
    copy (source.getptr());
    return *this;
}
```

Первая инструкция тела функции оператора присваивания выполняет копирование строки параметра в строку самого объекта (см. также раздел «Достоинства классов» в главе «Классы»). Вторая инструкция возвращает сам объект, то есть результат присваивания.

Так как функция короткая, можно ее определение включить в объявление класса, тогда фрагмент, объявляющий функцию, будет выглядеть так:


```
class MyStr
{
//...
    MyStr& operator=(const MyStr &source)
    {
        copy (source.getptr());
        return *this;
    }
};
```

Чтобы понять, как работает этот фрагмент программы, напомним, что выражение `str1=str2` транслируется в вызов функции `str1.operator=(str2)`, и сделаем некоторые пояснения.

Указатель `this` и его использование при создании функций

Внутри функции-члена ключевое слово `this` является указателем на текущий объект (под текущим объектом понимается объект, из которого была вызвана функция). Каждый раз, когда вызывается функция-член класса, к которому относится объект, C++ формирует скрытый указатель `this` как указатель на текущий объект.

Внутри определения функции-члена класса ссылки на данные-члены автоматически транслируются в ссылки на члены объекта с помощью указателя `this`. Во многих функциях-членах никогда не задают указатель `this` в явном виде, так как его использование подразумевается. В частности, при обращении к данным-членам `pstr` и `lstr` объектов класса `MyStr` подразумевается обращение `this -> pstr` и `this -> lstr` соответственно.

Иногда ссылка на сам объект нужна в явном виде, и тогда использование указателя `this` становится необходимым. Так происходит и в функции оператора присваивания.

Операторы присваивания C и C++ формируют результат, совпадающий с левым операндом (см. также раздел «Операции, выражения и комментарии» в главе «Основные

особенности программирования на C++»). Это свойство делает допустимыми следующие инструкции:

```
a=b=c=d=1; //int a,b,c,d;  
str1=str2=str3=""; //MyStr str1,str2,str3;
```

Для обеспечения этой возможности функция оператора присваивания должна возвращать значение левого операнда в своей последней инструкции.

Напомним еще раз, что выражение `str1=str2` транслируется в вызов функции `str1.operator=(str2)`, здесь `operator=` — имя функции. При переходе к ее выполнению текущим объектом становится `str1`. Следовательно, последней задачей функции оператора присваивания является возвращение самого объекта `str1`, то есть выполнение инструкции `return *this;`

Следует отметить преимущества задания типов ссылки при определении оператора присваивания. Ссылки позволяют выполнять передачу данных между объектами без создания лишних копий и без использования синтаксиса указателей, что могло бы привести к ошибкам при выполнении программы. В то же время возвращаемым значением функции оператора присваивания является ссылка на левый операнд, что делает возможным выполнение присваивания какого-либо значения сразу нескольким переменным.

Создание функции преобразования

Класс `MyStr`, который создавался в течение нескольких последних глав, стал довольно гибким и удобным. На данном этапе в нем определены конкатенация (объединение) и присваивание строк с более простым и удобным, по сравнению с первоначальным, синтаксисом — для использования этих операций достаточно написать:

```
MyStr str1="привет", str2("-"), str3;  
str3 = str1;  
str3 = str1 + str2 + str3;
```

В результате в переменной `str3` сформируется строка "привет-привет".

Однако для использования этого класса вместо типа `char*` нужно сделать так, чтобы функции ввода/вывода строк в C++, например `puts`, могли работать с параметрами соответствующего типа. Проще всего добиться этой цели, написав функцию преобразования объекта класса `MyStr` в тип `char*`. В разделе «Перегрузка конструкторов» в главе «Конструкторы» уже упоминалось о том, что конструктор тоже выполняет преобразование типов. Примером такого преобразования типов конструктором может служить инструкция `MyStr::MyStr(char *wstr)`.

Ответ на вопрос: что должен вызывать компилятор для преобразования типов, функцию или конструктор? — зависит от направления преобразования. Если выполняется преобразование из какого-либо типа в объект класса, то используется конструктор класса. Если преобразование производится из объекта какого-то класса к другому типу, то вызывается функция преобразования, являющаяся членом исходного класса.

Объявление функции преобразования не имеет возвращаемого типа и параметров. Синтаксис ее объявления выглядит так:

```
operator <тип> ();
```

В нашем случае для преобразования объекта класса `MyStr` в тип `char*` достаточно выделить указатель строки объекта. Так как эта функция достаточно коротка, включим ее определение в определение класса:

```
class MyStr
{
    //...
    operator const char* ()
    { return getptr(); }
};
```

Чтобы запретить модификацию этой строки, можно использовать ключевое слово `const`. Модификацию строки можно не запрещать, но при этом надо позаботиться о сохранности данных.

Обобщение класса MyStr

На протяжении трех глав с помощью многих ключевых понятий C++ мы последовательно создавали класс MyStr. Стоило ли уделять этому столько времени? Возможно, нет, если этот класс будет задействован лишь однажды. Однако, чтобы использовать его в других программах и сделать эффективным расширением C++, нужно выполнить следующую процедуру.

1. Включить файл заголовков `mystr.h` во все исходные файлы, которые используют класс MyStr.
2. Включить в проект задачи файл `mystr.obj`, полученный в результате компиляции файла `mystr.cpp`, содержащего все определения функций-членов класса MyStr. Файл `mystr.obj` может быть также включен в стандартную библиотеку, и тогда проект задачи должен содержать имя этой библиотеки.

Итак, файл заголовков `mystr.h` содержит следующее определение класса MyStr:

```
class MyStr
{
    char *pstr;
    int lstr;
public:
    //Конструкторы
    MyStr ();
    MyStr (char *wstr);
    MyStr (const MyStr &wstr);
    //Деструктор
    ~MyStr ();
    //Функции-члены класса
    char *getptr(void) const {return pstr;}
    int length(void) const {return lstr;}
    void copy(char *s);
    void plus(char *s);
    //Функции оператора сложения
    friend MyStr operator+(MyStr wstr1, MyStr wstr2);
```



```

    friend MyStr operator+(MyStr wstr1, char *wstr2);
    friend MyStr operator+(char *wstr1, MyStr wstr2);
//Функция оператора присваивания
    MyStr& operator=(const MyStr &source)
    {
        copy (source.getptr());
        return *this;
    }
//Функция преобразования типа
    operator const char* ()
    {
        return getptr();
    }
};

```

Отметим, что определения четырех функций-членов включены в объявление класса, и их тела во время компиляции будут встроены в места вызова этих функций (см. также раздел «Определение встраиваемой функции» в главе «Классы»).

Файл определений `mystr.cpp` функций-членов класса `MyStr`:

```

#include <string.h>
#include <mystr.h>

//Конструктор по умолчанию
MyStr::MyStr()
{
    pstr = new char[1];
    *pstr='\0';
    lstr=0;
}
//Конструктор, создающий объект из строки
MyStr::MyStr(char *wstr)
{
    pstr = new char[strlen(wstr)+1];
    strcpy (pstr, wstr);
    lstr=strlen(wstr);
}

```



```
//Конструктор копирования
MyStr::MyStr(const MyStr &wstr)
{
    char *tmpstr;
    int l;
    tmpstr = wstr.getptr();
    l = wstr.length();
    pstr = new char[l+1];
    strcpy (pstr, tmpstr);
    lstr=l;
}
//Деструктор
MyStr::~MyStr()
{
    delete [] pstr;
}
//Копирование параметра в область данных
void MyStr::copy(char *s)
{
    int n;
    n = strlen(s);
    if ( lstr != n)
    {
        if ( pstr ) //если pstr - не пустая строка
            delete [] pstr;
        pstr=new char[n+1]; //длина строки+1
        lstr=n;
    }
    strcpy(pstr, s);
}
//Конкатенация
void MyStr::plus(char *s)
{
    int n;
    char *ptmp;
    n = strlen (s);
    if (n==0) return;
    ptmp = new char[lstr+n+1]; //длина строки+1
    if (pstr) //если pstr - не пустая строка
```



```

    {
        strcpy (ptmp, pstr);
        delete [] pstr;
    }
    strcat(ptmp, s);
    pstr=ptmp;
    lstr+=n;
}

//Функции оператора сложения
MyStr operator+(MyStr wstr1, MyStr wstr2)
{
    MyStr tmpstr(wstr1);
    tmpstr.plus(wstr2.getptr());
    return tmpstr;
}
MyStr operator+(MyStr wstr1, char *wstr2)
{
    MyStr tmpstr(wstr1);
    tmpstr.plus(wstr2);
    return tmpstr;
}
MyStr operator+(char *wstr1, MyStr *wstr2)
{
    MyStr tmpstr(wstr1);
    tmpstr.plus(wstr2.getptr());
    return tmpstr;
}

```

Еще один пример перегрузки операторов

Прежде чем завершить обсуждение перегрузки операторов, рассмотрим следующий пример: дополним функциями операторов сложения и присваивания класс `GraphPoint`, введенный в примере 21 главы «Классы» и дополненный в примере 30 главы «Конструкторы».

```

class GraphPoint
{
    private:
        int x, y, color;

```



```
public:
//конструкторы
    GraphPoint ();
    GraphPoint (int wx, int wy, int wcolor);
    GraphPoint (const GraphPoint &wgp);
//функции
    int getcolor(void);
    void setnewcolor(int wcolor);
//операторы
    GraphPoint operator+(int dy);
    GraphPoint& operator=(const GraphPoint &wgp);
};
```

 **ПРИМЕЧАНИЕ** Конструктор копирования, объявленный в классе GraphPoint, выполняет те же действия, что и конструктор копирования по умолчанию, поэтому описание и определение его для этого класса можно опустить, в отличие от конструктора по умолчанию.

При определении операторов для работы с объектами класса важно решить, какие операторы следует поддерживать. В случае с классом MyStr операцию сложения двух строк можно было определить как создание третьей строки. В случае класса GraphPoint необходимость сложения двух графических точек менее очевидна, но можно определить оператор сложения как изменение координаты y:

```
GraphPoint GraphPoint::operator+(int dy)
{
    GraphPoint wgp (*this);
    wgp.y += dy;
    return wgp;
}
```

Определим оператор присваивания как копирование всех данных-членов одного объекта в другой. В данном случае эта функция не является необходимой, так как она выполняет присваивание тем же способом, что и функция оператора присваивания, определяемая компилятором по умолчанию:


```
GraphPoint&
GraphPoint::operator=(const GraphPoint & wgp)
{
    x = wgp.x;
    y = wgp.y;
    color = wgp.color;
    return *this;
}
```

Данные-члены правого операнда оператора присваивания копируются в данные-члены левого операнда, то есть объекта, из которого вызывается функция (см. также раздел «Создание функции оператора присваивания»). Затем текущий объект с измененными данными-членами возвращается в вызывающую функцию.

Обобщение определения операторов

Мы привели примеры функций операторов сложения, присваивания и функции преобразования. Помимо них существует еще несколько групп перегружаемых операторов.

Имя функции оператора

Как уже говорилось, для любого оператора @ имя его функции определяется как

operator@

Исключением является имя функции преобразования класса в тип. Ее имя определяется как

operator <тип> ()

Бинарные операторы

Когда компилятор встречает выражение `object1@object2`, где @ — оператор, `object1` и `object2` — произвольные переменные, оно транслируется в вызов функции `object1.operator@(object2)`, если такая функция определена как функция-член класса, к которому относится объект `object1`.

Если функция не определена внутри этого класса, выражение `object1@object2` транслируется в вызов функции

`operator@(object1, object2)`. Для доступа к членам класса функция `operator@` должна быть объявлена как функция-друг этого класса с помощью ключевого слова `friend`.

Преимущество использования функций-друзей заключается в том, что они делают возможным выполнение одной и той же операции независимо от того, где расположен операнд `object1`, слева или справа от оператора. Для поддержки последнего варианта (`object1` находится справа от оператора) достаточно написать отдельную функцию:

```
operator@(object2, object1)
```

Если `object2` имеет стандартный тип C++, например, `int` или `char*`, использование функций-друзей является единственным способом определения такой операции.

Унарные операторы

Когда компилятор встречает выражение `@object1`, где `object1` — произвольная переменная, он транслирует это выражение в вызов функции `object1.operator@()`, если такая функция определена как функция-член класса, к которому относится объект `object1`.

Если функция определена вне класса, выражение `@object1` транслируется в вызов функции `operator@(object1)`. Для доступа к членам класса функция `operator@` объявляется как функция-друг этого класса с помощью ключевого слова `friend`.

В качестве примера определим унарную операцию минус (`-`) для класса `GraphPoint`, как инверсию цвета графической точки.

```
class GraphPoint
{
    //...
    GraphPoint operator-();
};
GraphPoint GraphPoint::operator-()
{
    GraphPoint wgp (*this);
    wgp.color = 15-wgp.color;
    return wgp;
}
```


Как и во всех функциях операторов, кроме функции преобразования, наличие инструкции `return` обязательно. Однако класс объекта, возвращаемого им и функцией, может не совпадать с классом `GraphPoint`. Результатом этой операции может быть объект любого другого класса.

Оператор присваивания

Оператор присваивания определяет, каким образом должно выполняться присваивание одного объекта другому объекту того же класса. Хотя присваивание является бинарной операцией, во многом похожей на все другие бинарные операции, она имеет некоторые ограничения. Функция оператора присваивания должна быть определена как функция-член класса следующим образом:

```
<имя класса> & operator=(const <имя класса> &<объект класса>)
```

Функция вызывается как функция-член класса объекта, являющегося левым операндом с правым операндом в качестве параметра. Последней строкой в функции оператора присваивания должно быть возвращение измененного левого операнда, выполняемое через указатель `this`:

```
return *this;
```

Присваивание переменных разных типов

Оператор присваивания определяет, каким образом должно выполняться присваивание одного объекта другому объекту того же класса. Не существует правильных функций операторов присваивания между переменными разных типов. Вместо них компилятор вызывает подходящий конструктор или функцию преобразования, если они существуют.

```
MyStr str1="Привет",str2; //конструктор для str1  
str2 = str1; //оператор присваивания  
char*str3 = str1; //функция преобразования
```



ПРИМЕЧАНИЕ Чтобы последняя инструкция могла выполняться, следует убрать ключевое слово `const` в объявлении и определении функции преобразования типа класса `MyStr`.

Специальные операторы присваивания

Определение операторов сложения (+), вычитания (-) и присваивания (=) еще не означает определения операторов присваивания со сложением (+=) и присваивания с вычитанием (-=). Все эти операторы (+=, -=, *= и т. д.) являются самостоятельными, и определение функции для каждого из них должно быть выполнено отдельно.

Например, можно определить функцию оператора присваивания со сложением для класса `MyStr` следующим образом:

```
class MyStr
{
//...
    MyStr& operator+=(const MyStr &wstr);
};
MyStr& MyStr::operator+=(const MyStr &wstr)
{
    plus (wstr.getptr());
    return *this;
}
```

Благодаря краткости этой функции ее тело можно включить в объявление класса:

```
class MyStr
{
//...
    MyStr& operator+=(const MyStr &wstr)
    {
        plus (wstr.getptr());
        return *this;
    }
};
```

Как и в функциях простых операторов присваивания, последней инструкцией функций специальных операторов присваивания должна быть инструкция `return *this;`.

Операторы инкремента и декремента

Несмотря на то что операторы инкремента (++) и декремента (--) являются унарными операторами и подчиняются тем же правилам, что и все унарные операторы, они обла-

дают одной особенностью: могут быть префиксными и постфиксными.

Префиксный вариант сначала изменяет операнд, а затем возвращает его значение. Функция возвращает ссылку на текущий объект.

Постфиксный вариант возвращает копию старого значения объекта. Этот вариант имеет вспомогательный параметр `wint` типа `int`, который передает значение ноль (0) во время вызова. Единственным назначением этого параметра является создание двух различных определений функций для префиксного и постфиксного вариантов.

Следующий пример иллюстрирует постфиксный и префиксный варианты оператора инкремента (`++`).

Пример 37

```
class GraphPoint
{
    //...
    GraphPoint& operator++(); //префиксный вариант
    GraphPoint operator++(int); //постфиксный вариант
};
//префиксный вариант
GraphPoint& GraphPoint::operator++()
{
    y++;
    return *this;
}
//постфиксный вариант, который включает
//вспомогательную переменную wint
GraphPoint GraphPoint::operator++(int wint)
{
    GraphPoint tmpgp (*this);
    y++;
    return tmpgp;
}
```

Оператор выделения элемента массива

Оператор выделения элемента массива (`[]`) нужен при определении и использовании класса, являющегося массивом.

вом. Функция оператора выделения элемента массива получает один параметр целого типа, определяющий индекс в массиве (как обычно в C++, индексы начинаются с 0).

Функция должна возвращать ссылку, чтобы обеспечить возможность присутствия индексов в выражениях по обе стороны оператора присваивания (=).

Операторы new и delete

Эти операторы применяются для распределения памяти в классе. Чтобы написать функции этих операторов, в программу включается файл заголовков `stddef.h` и используются следующие инструкции:

```
#include <stddef.h>
class NewClass
{
    ...
    void *operator new(size_t);
    void operator delete(void*, size_t);
};
```

Функция оператора `new` всегда возвращает указатель типа `void*`, а оператор `delete` всегда получает параметр типа `void*` (указатели будут автоматически приводиться к нужному типу). Функция оператора `new` может включать дополнительные параметры, которые соответствуют одному или более значениям, используемым для инициализации объекта. В зависимости от количества и типа параметров эта функция может перегружаться.

Чаще всего функции операторов `new` и `delete` не нуждаются в новых определениях, так как стандартные определения этих функций вполне удовлетворяют всем требованиям.

Оператор вызова функции

Целью оператора вызова функции `(( ))` является обеспечение возможности указания имен объектов как имен функций. Можно определить функцию этого оператора несколько раз. В зависимости от количества и типа параметров она будет перегружаться.

В следующем примере объект `myprn` может быть использован так, как если бы он был функцией.

Пример 38

```
class prn_class
{public:
    int operator()(long lp);
    int operator()(char *str, int n);
    //...
};
prn_class myprn;
myprn ("привет", 3);
//вызов функции
//prn_class::operator()(char *str, int n);
```

Ограничения на перегрузку операторов

Ниже перечислены важные ограничения, о которых нужно помнить при перегрузке операторов.

- Можно перегружать только те операторы, которые существуют в C++. Нельзя перегружать операторы доступа к членам классов (`.`), области действия (`::`), указателя на член (`.*`) и условный (`?:`).
- Независимо от классов, в которых они определены, операторы подчиняются стандартным правилам приоритета (порядок их выполнения в выражениях сохраняется).
- Все функции операторов, определенные для классов, наследуются классами-потомками, кроме функции оператора присваивания.
- Функции для каждого оператора должны быть определены отдельно, в том числе для операторов типа `+=`, `-=`, `*=` и им подобных.
- Значения параметров по умолчанию не могут использоваться в функциях операторов.
- Не допускается определение функций операторов в виде статических функций-членов (определяемых с помощью ключевого слова `static`).

Наследование

Наследование — удобное средство C++, позволяющее избежать переписывания функций и данных из одного класса в другой. Наследование позволяет переходить от классов, обладающих общими свойствами, к все более и более специализированным. Достаточно определить класс производным от другого (базового) класса, объявить и определить в нем новые члены, и он, помимо новых, унаследует все члены базового класса. Использование производных классов позволяет строить целые иерархии классов.

Дилемма программного обеспечения

В предыдущих главах этой книги был построен класс `MyStr`. Чтобы воспользоваться им, не вникая в детали реализации его функций и операторов, достаточно иметь файл заголовков, содержащий объявление класса, и объектный (с расширением `obj`) или библиотечный файл, содержащий определение всех функций класса в скомпилированной форме.

Именно в таком виде, без исходных текстов функций, программист чаще всего использует классы и функции, определенные и написанные не им самим. Но что делать, если нужно модифицировать такой класс?

Чтобы решить эту проблему, можно определить этот класс, допустим `MyStr`, заново. Более удачным решением является создание нового класса. При таком подходе достаточно просто объявить в качестве членов класса объект класса `MyStr` и недостающие данные и функции.

Однако лучшим вариантом по объему требуемой работы является наследование. Это означает создание нового класса, *производного* (*derived*) от класса `MyStr` (последний в

этом случае называется *базовым*). При наследовании все члены базового класса (в нашем случае `MyStr`) становятся полноправными членами производного класса.

Создание производного класса

Окончательная версия класса `MyStr`, приведенная в разделе «Обобщение класса `MyStr`» главы «Перегрузка операторов», поддерживает несколько полезных функций и операций, перечисленных в табл. 26.

Таблица 26. Функции и операторы класса `MyStr`

Функция или оператор	Описание
<code>Getptr</code>	Возвращает указатель на строку данных, заканчивающуюся нуль-терминатором
<code>Length</code>	Возвращает длину строки данных
<code>Copy</code>	Копирует в строку данных содержимое параметра типа <code>char*</code>
<code>Plus</code>	Дописывает содержимое параметра типа <code>char*</code> в текущую строку данных
<code>+</code>	Записывает две строки в одну друг за другом. По крайней мере, одна из строк должна иметь тип <code>MyStr</code> . Другая может иметь тип <code>MyStr</code> или <code>char*</code>
<code>=</code>	Копирует строки данных из другого объекта класса <code>MyStr</code>

Класс также поддерживает несколько полезных конструкторов. Сохраним все эти функции и добавим еще две:

- функцию `output`, позволяющую выполнять вывод строки данных на стандартное устройство;

- функцию `input`, позволяющую выполнять ввод строки со стандартного устройства ввода.

Создание производного класса путем наследования делает простой процедуру добавления этих функций, даже если нет доступа к тексту функций базового класса.



ПРИМЕЧАНИЕ В некоторых источниках базовый класс называют родительским классом или предком, а производный класс — дочерним или потомком.

Класс `NewClass`, производный от базового класса `BaseClass`, объявляется следующим образом:

```
class NewClass: public BaseClass
{
<объявления>
};
```

Такое объявление приводит к тому, что класс `NewClass`, помимо своих данных и функций, содержащихся в объявлениях, автоматически содержит все данные и функции базового класса `BaseClass`. Программист может изменять и дополнять их в соответствии со своими потребностями, используя для этого операторы доступа к членам объекта `(.)` или `(->)`.

В данном контексте ключевое слово `public` является спецификатором доступа и определяет механизм доступа к членам базового класса со стороны производного. Кроме ключевого слова `public` в описании доступа можно использовать ключевые слова `private` и `protected` (см. также описание этих слов в главе «Ключевые слова»), однако вариант, приведенный первым (`public`), является наиболее предпочтительным.



ВНИМАНИЕ Прежде чем использовать какой-либо класс в качестве базового, он должен быть полностью определен. Чаще всего для этого достаточно включить в программу соответствующий файл заголовков с помощью директивы `include`.

Приведем пример класса `MyIOStr`, производного от базового класса `MyStr`.

Пример 39

```
#include <mystr.h>
class MyIOStr : public MyStr
{public:
    void input(void);
    void output(void);
};
```

Как только класс объявлен, его можно использовать для определения объектов. Эти объекты имеют доступ ко всем членам базового класса, а также к функциям `input` и `output`.

Как обычно, рекомендуется определение класса записать в отдельный файл заголовков, например `myiostr.h`. Фрагмент программы с этим классом может выглядеть следующим образом.

Пример 39

(продолжение)

```
#include <myiostr.h>
MyIOStr str1, str2, str3;
str1.copy ("Введите строку");
str2.plus(":");
str2.output();
str3.input();
str2.copy("Введена строка");
str2.plus (str3.getptr());
str2.output();
```

Определение функций-членов производного класса `MyIOStr` может быть выполнено двумя способами так же, как обычное определение функций-членов класса. В первом случае, при объявлении функции внутри объявления класса, компилятор встраивает тело функции в месте ее вызова. Во втором случае, при объявлении функции вне объявления класса, перед именем функции должно быть записано имя класса с двумя двоеточиями (`::`).

Пример 39**(продолжение)**

```
//определения новых функций класса MyIOStr
#include <stdio.h>
#include "myiostr.h"
//файл myiostr.h включает в себя файл mystr.h
void MyIOStr::output (void)
{
    printf ("%s", getptr());
}
void MyIOStr::input (void)
{
    char buf[256];
    gets (buf);
    copy (buf);
}
```

Здесь определены две функции нового класса. Обе они имеют префикс `MyIOStr::`. Остальные функции унаследованы от базового класса `MyStr`, определены в файле `mystr.cpp` и имеют префикс `MyStr::`. Например, определение функции копирования, приведенное в разделе «Обобщение класса `MyStr`» главы «Перегрузка операторов», выглядит так:

```
void MyStr::copy(char *s)
{
    //...
}
```

Таким образом, класс `MyIOStr` поддерживает два вида функций: функции базового класса `MyStr` и функции самого класса `MyIOStr`. Префикс, используемый в определении, меняется в зависимости от того, где эти функции объявлены и определены: в базовом или производном классе.

Последней инструкцией в определении функции `MyIOStr::input(void)` примера 39 является вызов функции копирования `MyStr::copy`. Благодаря свойствам наследования она также является функцией-членом класса `MyIOStr`, поэтому обращение к ней выполняется как к обычной функции класса `MyIOStr`.

Ситуация с использованием производных классов значительно усложняется, если в производном классе необхо-

димо изменение одной или нескольких функций базового класса.

Пример 40

```
class MyIOStr : public MyStr
{public:
    void input(void);
    void output(void);
    void copy(char *s);
//новое объявление функции копирования
};
void MyIOStr::copy (char *s)
{//новое определение функции копирования
}
```

В результате переопределения функции `copy` в производном классе получилось две функции, различные по исполняемым действиям, но имеющие одинаковые имена: одна, определенная в классе `MyIOStr`, и другая, определенная в классе `MyStr`. Обе они могут быть вызваны из функций-членов класса `MyIOStr`. Но теперь для вызова первоначальной версии функции `copy` (из базового класса) мы должны использовать оператор уточнения области действия (`::`), то есть вызывать эту функцию как `MyStr::copy`. В противном случае будет вызываться функция `MyIOStr::copy`.

```
void MyIOStr::input (void)
{
    char buf[256];
    gets ( buf);
    MyStr::copy (buf);
//вызов функции copy базового класса
}
```

В целом, когда компилятор встречает имя функции, он определяет область определения этого имени в следующем порядке.

1. Если имя находится внутри определения функции-члена класса, компилятор проверяет, есть ли объявление этого

имени внутри данного класса. Если такое объявление есть, выбирается член данного класса.

2. Компилятор проверяет объявления базового класса. Если базовый класс тоже является производным, то проверка выполняется для его базового класса. Как только объявление найдено в каком-либо базовом классе, выбирается член этого класса.
3. Если имя не объявлено ни в одном из классов, компилятор проверяет, не объявлена ли функция глобальной.

 **ПРИМЕЧАНИЕ** Если предполагается, что функция может быть переопределена для производного класса, ее следует объявить *виртуальной* (*virtual*). Подробнее об этом будет рассказано в конце этой главы.

Иерархия наследования

Подобно базовым классам, которые могут иметь свои собственные базовые классы, производные классы также могут иметь свои производные классы.

Для примера объявим класс `MyFontIOStr` производным от класса `MyIOStr`.

Пример 41

```
#include <mystr.h>
class MyFontIOStr : public MyIOStr
{
    int ptsize;
public:
    void clr(void){copy("");}
    void setchar(int c, int n);
    void setfont(int size);
};
```

Новый класс `MyFontIOStr` наследует все члены классов `MyIOStr` и `MyStr`, так как каждое последовательное наследование добавляет или переписывает члены базового класса. Ни один из членов не исчезает, хотя права доступа могут иногда изменяться.

В классе `MyFontIOStr` появились новые члены, которых нет у базовых классов, — это переменная `ptsize` и три новых функции. Кроме того, он имеет три функции-члена, унаследованных от класса `MyIOStr`, плюс два данных-члена и более десятка функций-членов, унаследованных от класса `MyStr`.

Эти три класса, `MyFontIOStr`, `MyIOStr` и `MyStr`, создают иерархию классов, показанную на рис. 1.

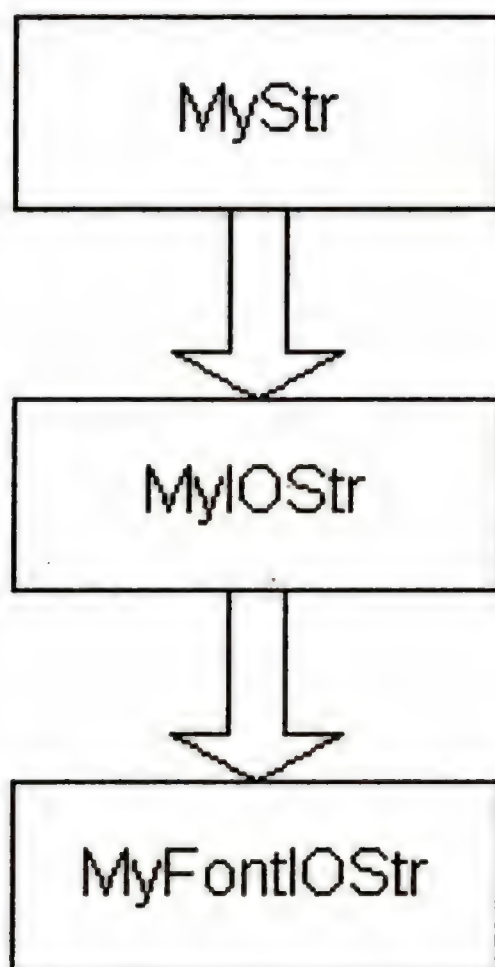


Рис. 1. Иерархия классов `MyFontIOStr`, `MyIOStr` и `MyStr`

Исходный класс `MyStr` не является непосредственным базовым классом класса `MyFontIOStr`, самого младшего в этой иерархии. Но через наследование он передает свои члены этому классу так же, как родители родителей передают свои гены внукам.

Возможно создание значительно более сложных иерархий классов.

Создание класса без наследования

Использование наследования не является единственным методом создания многократно используемого программ-

ного обеспечения. Более того, в некоторых случаях этот метод не является оптимальным. Посмотрим, тем не менее, как без наследования можно сократить объем работы программиста.

Предположим, программист не имеет доступа к исходным текстам функций-членов класса `MyStr`, за исключением объявления этого класса в файле заголовков, но ему необходимо создать класс `MyIOStr`, обладающий, помимо своих, всеми возможностями класса `MyStr`. Как создать такой класс без переписывания всех функций класса `MyStr` и без наследования?

Одним из способов решения проблемы является создание класса, представляющего из себя оболочку вокруг объекта класса `MyStr`, и подстановка вместо вызовов функций-членов нового класса вызовов функций-членов класса `MyStr`.

Пример 42

```
#include "mystr.h"
class MyIOStr : public MyStr
{
    MyStr wstr;
    public:
//новые функции класса
    void input(void);
    void output(void);
//старые функции, имитирующие наследование
    char *getptr(void) const {return wstr.getptr();}
    int length(void) const {return wstr.length();}
    void copy(char *s){ wstr.copy(s);}
    void plus(char *s){ wstr.plus(s);}
//...
};
```

К недостаткам этого метода относится то, что каждый вызов функции-члена нового класса будет транслироваться в вызов соответствующей функции-члена класса `MyStr`. Таким образом, кроме вышеперечисленных функций, должны быть созданы все функции, транслирующие вызов функ-

ций операторов, вызов функции преобразования и вызов конструкторов класса `MyStr`.

Сравним это объявление класса с объявлением класса `MyIOStr`, приведенном в примере 39.

```
#include <mystr.h>
class MyIOStr : public MyStr
{
    public:
        void input(void);
        void output(void);
};
```

Оба класса, объявленные столь различными способами, содержат одни и те же члены, и работа с ними будет выполняться абсолютно одинаково. Также очевидно, что последнее объявление класса `MyIOStr` намного проще, понятнее и элегантнее.

Можно попытаться создать новый класс как оболочку объекта класса `MyStr` еще одним способом. Для этого вместо повторного объявления функций-членов класса `MyStr` можно использовать их вызов через объект этого класса, являющийся членом нового класса. В этом случае объявление класса будет значительно короче, чем в примере 42.

```
#include "mystr.h"
class MyIOStr
{
    public:
        MyStr wstr;
    //только новые функции класса
        void input(void);
        void output(void);
};
```

Однако функции класса `MyStr` нам придется вызывать как функции объекта `wstr`, и соответствующий фрагмент программы будет выглядеть так:

```
MyIOStr IOStr;
//...
IOStr.wstr.copy("Привет!");
```


Заметим, что при необходимости создания класса, подобного классу `MyFontIOStr` из примера 41, процедура вызова функции `сору` еще более усложнится. Кроме того, объект `wstr` должен быть объявлен как `public` или `protected`, иначе компилятор выдаст сообщение о невозможности доступа к нему.

Наследование и оболочка объекта

Теоретически, можно рассматривать создание производного класса и создание класса в виде оболочки вокруг объекта, как совершенно разные методы.

Первый метод, согласно теории объектно-ориентированного программирования, следует применять в ситуации, когда производный класс `A` является разновидностью (частным случаем) базового класса `B`. Опираясь на такую трактовку наследования, можно сказать, что класс `MyIOStr` является разновидностью класса `MyStr`, поэтому к нему добавляются новые функции. Тем не менее оба класса остаются строками.

При применении второго метода один класс использует объект другого класса. Например, можно создать класс `Person`, содержащий несколько объектов другого класса (`MyStr`):

```
class Person
{
    MyStr surname;
    MyStr name;
    MyStr patronymic;
    int age;
};
```

В данном случае нельзя создавать класс `Person` как разновидность класса `MyStr`, так как он имеет совершенно другой смысл.

Различие методов создания этих двух классов (`Person` и `MyIOStr`) очевидно. Однако в некоторых случаях сделать выбор нужного метода для создания класса довольно трудно.

Виды доступа: **public**, **private** и **protected**

C++ имеет три вида спецификатора доступа к членам класса: **public**, **private** и **protected**.

Внутри объявления класса **private** является спецификатором доступа к членам этого класса по умолчанию, поэтому обычно при объявлении класса он опускается.

Ключевое слово **protected** определяет уровень доступа, промежуточный между **public** и **private**. Члены класса, объявленные таким образом, доступны в функциях-членах этого класса и классов, производных от него, но не видны вне этих областей.

Обратимся снова к классу `MyIOStr`, объявленному в примере 39.

```
#include <mystr.h>
class MyIOStr : public MyStr
{
    public:
        void input(void);
        void output(void);
};
```

Этот класс является производным базового класса `MyStr`, а значит, все члены класса `MyStr`, в том числе `pstr` и `lstr`, являются членами класса `MyIOStr`. Однако попытка определить функцию `display`, выполняющую вывод на экран строки `pstr`, приведет к ошибке, так как строка `pstr` объявлена по умолчанию со спецификатором доступа **private** и доступ к ней вне функций-членов класса `MyStr` невозможен.

```
void MyIOStr::display(void)
{
    printf ("%s", pstr); //Ошибка! Нет доступа к pstr
};
```

Чтобы избежать этой ошибки, следует воспользоваться функциями `getptr` или `copy`.

Другим возможным способом решения этой проблемы является объявление данных-членов класса `MyStr` `pstr` и `lstr`

со спецификатором доступа `protected`. При таком объявлении они становятся доступными не только для функций-членов класса `MyStr`, но и для функций-членов класса `MyIOStr`.

В некоторых случаях объявление данных-членов с помощью спецификатора `protected` является необходимым. Такое объявление обеспечивает максимальную гибкость при использовании данных-членов базового класса в функциях-членах производного класса.

Однако в приведенном примере можно оставить переменные `pstr` и `lstr` закрытыми (`private`), так как функции `getptr`, `length`, `copy` и `plus` обеспечивают полный доступ к данным и позволяют избежать возможных ошибок.

Наконец, данные-члены могут быть описаны со спецификатором `public`. Это делает их доступными для любых функций, но лишает программиста одного из важных преимуществ работы с классами: возможности переопределения функций класса без необходимости изменения использующих их программ (см. также раздел «Достоинства классов» в главе «Классы»).

Пример создания класса

Как уже было сказано, основной идеей наследования является возможность объявить базовый класс, а потом создавать на его основе любое количество производных классов. Эти классы, в свою очередь, могут быть базовыми для других классов, таким образом, иерархическое дерево классов может быть сколь угодно высоким и разветвленным.

Определим базовый класс для хранения информации о книгах, который содержит следующие данные: название, имя автора, издательство, год и место издания.

Пример 43

```
//файл mybook.h
#include "mystr.h"
class MyBook
```


Пример 43

(продолжение)

```
{
    public:
        MyStr title;
        MyStr author;
        MyStr publisher;
        int year;
        MyStr place;
//конструкторы
    MyBook() {};
    MyBook (MyStr wtitle, MyStr wauthor, MyStr wpublish,
            int wyear, MyStr wplace);
};
//-----
//файл mybook.cpp
#include "mystr.h"
MyBook::MyBook (MyStr wtitle, MyStr wauthor,
                MyStr wpublish, int wyear, MyStr wplace)
{
    title = wtitle;
    author = wauthor;
    publisher = wpublish;
    year = wyear;
    place = wplace;
}
```



ПРИМЕЧАНИЕ Вместо определения типа полей title, author, publisher и place как объектов класса MyStr можно определить их стандартным строковым типом C++ char*. В этом случае первую строку файла mybook.h следует заменить строкой #include <string.h>, а вместо присваивания должна использоваться функция strcpy, копирующая строки C++.

Такое определение подходит для любой книги. Однако для литературы по искусству хотелось бы хранить некоторые дополнительные сведения, например, область искусства, к которой относится книга (архитектура, живопись, кино, театр). Определим класс MyArtBook как производный от класса MyBook.


```
class MyArtBook:public MyBook
{
    public:
        MyStr branch;
//конструктор
    MyArtBook () {};
};
```

Таким образом, книги по искусству определены как подмножество всех книг.

Подобным образом можно определить словари как подмножество всех книг, имеющих три дополнительных атрибута: язык оригинала, язык перевода и количество слов.

```
class MyDictBook:public MyBook
{
    public:
        MyStr lang1;
        MyStr lang2;
        int narticles;
//конструктор
    MyDictBook () {};
};
```

Итак, имеются два класса, производных от одного базового. Выделим подмножество первого из них: определим в качестве еще одного производного класса статьи в книгах по искусству кино. К данным-членам класса MyArtBook добавим еще два члена, определяющих название кинофильма и дату его выпуска на экран.

```
class MyCinemaArticle:public MyArtBook
{
    public:
        MyStr MTitle;
        int MYear;
};
```

Общая схема построенной иерархии классов представлена на рис. 2.

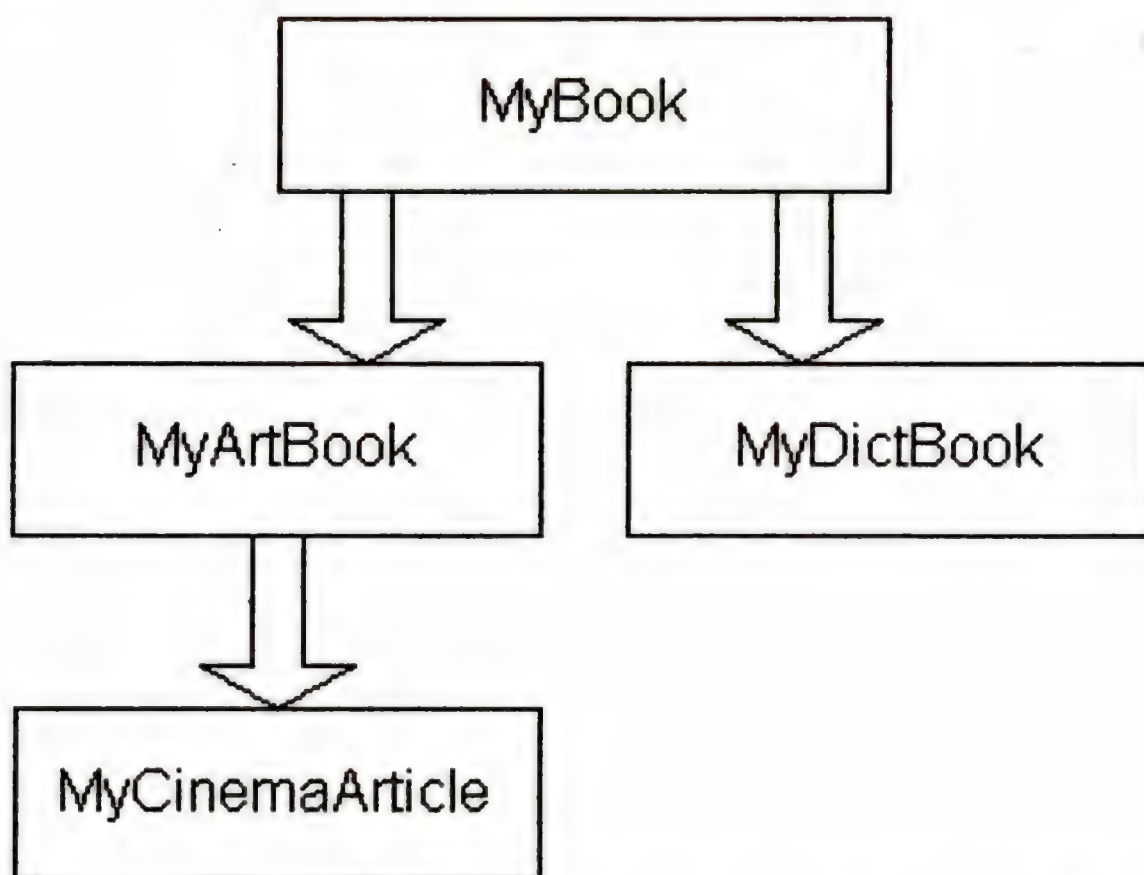


Рис. 2. Иерархия классов для хранения данных о книгах и фильмах

Конструкторы базовых классов

Наследование является средством включения в класс членов (как функций, так и данных) другого класса. Для инициализации этих унаследованных членов в С++ предусмотрены *конструкторы базового класса*.

В предыдущем разделе введен класс книг по искусству MyArtBook, который наследует члены класса MyBook и добавляет к нему свои данные — объект branch. Для такого объявления удобно иметь еще один конструктор для инициализации всех данных-членов, включая унаследованных от класса MyBook:

```
#include "mystr.h"
class MyArtBook:public MyBook
{
    public:
        MyStr branch;
//конструкторы
    MyArtBook () {};
    MyArtBook (MyStr wtitle, MyStr wauthor, MyStr wpublish,
               int wyear, MyStr wplace, MyStr wbranch);
};
```


Новый конструктор инициализирует все данные-члены:

```
#include "mystr.h"
MyArtBook::MyArtBook (MyStr wtitle, MyStr wauthor,
                      MyStr wpublish, int wyear,
                      MyStr wplace, MyStr wbranch)
{
    title = wtitle;
    author = wauthor;
    publisher = wpublish;
    year = wyear;
    place = wplace;
    branch = wbranch;
}
```

Однако приведенный вариант конструктора не является эффективным и приводит к возникновению двух проблем. В первую очередь, C++ всегда строит объект производного класса, сначала вызывая для него конструктор базового класса. Если такой конструктор не определен в явном виде, компилятор вызывает конструктор по умолчанию. Таким образом, члены класса MyArtBook инициализируются дважды: первый раз неявно конструктором базового класса MyBook, а затем явно конструктором класса MyArtBook. Вторая проблема значительно существеннее. В примере все члены класса MyBook объявлены как public, поскольку в случае их объявления как private конструктор MyArtBook::MyArtBook не будет иметь к ним доступа.

Решением этих проблем является запись соответствующего конструктора базового класса в определении конструктора MyArtBook::MyArtBook:

```
class::<производный класс>(<список параметров1>):
    <базовый класс>(<список параметров2>)
{<инструкции>}
```

При этом подразумевается следующее:

- <список параметров1> должен совпадать по числу и количеству параметров со списком параметров конструктора базового класса (<список параметров2>);
- любой параметр из списка <список параметров1> может входить в <список параметров2>.

Используя определение конструктора `MyArtBook::MyArtBook` через конструктор базового класса `MyBook::MyBook`, получим:

```
MyArtBook::MyArtBook (MyStr wtitle, MyStr wauthor,
                      MyStr wpublish, int wyear,
                      MyStr wplace, MyStr wbranch):
MyBook(wtitle, wauthor, wpublish, wyear, wplace)
{
    branch = wbranch;
}
```

Теперь четыре данные-члены, унаследованные от класса `MyBook`, инициализируются только один раз. Кроме того, конструктор класса `MyArtBook` стал значительно короче и не требует обращения к данным-членам класса `MyArtBook`.

Базовые классы и указатели

Последняя точка в проблеме наследования будет поставлена в следующей главе, посвященной виртуальным функциям. Многие программисты рассматривают эти функции как ядро объектно-ориентированного программирования, поэтому стоит уделить особое внимание проблемам, для решения которых они предназначены.

Наследование порождает некие отношения между типами в C++. Вообще говоря, C++ не допускает присваивания указателей разных типов без приведения типов. Однако присваивание объекта производного класса указателю базового класса возможно. Для определенных выше классов `MyStr`, `MyIOStr`, `MyFontIOStr` допустимы следующие инструкции:

```
MyStr pstr;
MyIOStr pIOstr;
MyFontIOStr pFontIOstr;
//...
pstr = pIOstr;
//...
pstr = pFontIOstr;
//...
pIOstr = pFontIOstr;
```


Интересно, что такое присваивание допускается только в одном направлении: указатель базового типа может указывать на производный тип. Это ограничение объясняется довольно просто.

Все члены базового класса наследуются объектами производного класса. Следующая инструкция работает правильно, независимо от того, является `pwstr` указателем на объект класса `MyStr` или `MyIOStr`:

```
pwstr->plus ("Вторая часть строки");
```

Но обратное присваивание не поддерживается, так как базовый класс является только подмножеством производного: часть членов, входящих в производный класс, не входит в базовый. Если `pwstr` — это указатель на объект класса `MyStr`, как будет выполняться следующая инструкция?

```
pwstr->input();
```

Понятно, что она вызовет ошибку уже при компиляции.

Переменной, объявленной как указатель на класс, можно присвоить адрес объекта этого класса или класса, производного от него. Даже если в программе используется показанный ниже фрагмент, значения всех трех указателей будут адресами объектов класса `MyStr`:

```
MyStr *pwstr[3];  
pwstr[0] = new MyStr;  
pwstr[1] = new MyIOStr;  
pwstr[2] = new MyFontIOStr;
```

В следующем фрагменте программы в соответствии с типом объекта, определенным при компиляции, последняя инструкция будет вызывать функцию `MyStr::copy`, но не `MyIOStr::copy`:

```
MyIOStr Iowstr;  
MyStr *pwstr = &Iowstr;  
pwstr->copy("Привет!");
```

По другому правилу выполняется определение вызываемой функции, если она объявлена *виртуальной* (`virtual`).

Для виртуальных функций определение варианта, который должен выполняться, производится во время выполнения программы, а не во время компиляции. Виртуальные функции используют *позднее связывание* (late binding), которое означает, что следующая ссылка на функцию определяет реальный адрес функции только во время выполнения программы:

```
pwstr->copy("Привет!");
```

Виртуальные функции и правила их использования описываются в следующей главе.

Виртуальные функции

Как было показано в предыдущей главе, иногда нужно вызывать функцию, не зная заранее класс объекта, для которого этот вызов будет выполняться. Фактически, класс этого объекта может даже не существовать на момент, когда пишется вызов функции: новые классы могут быть объявлены позже, соответственно, и их функции-члены также появятся позже. При этом не должно быть необходимости снова компилировать основную программу. Такое свойство делает программу децентрализованной, а решение, какую из функций вызывать, полностью передается самим объектам. Эту возможность предоставляют программисту *виртуальные функции*.



ПРИМЕЧАНИЕ Термин *виртуальный* (virtual) обычно описывает предмет, являющийся не совсем реальным, но работающий так, как будто он существует на самом деле. В нашем случае ключевое слово `virtual` имеет несколько иной смысл, так как виртуальные функции совершенно реальны. Просто в отличие от функций с фиксированным адресом, они имеют не прямой, а косвенный вызов через таблицу виртуальных функций (см. также раздел «Механизм вызова виртуальных функций»). Конкретный адрес этого вызова определяется только на этапе выполнения программы.

Начинающему программисту не всегда понятно, чем хорош такой подход. Достоинства виртуальных функций рассматриваются в разделе «Программирование команд меню».

Применение ключевого слова `virtual`

Чтобы превратить обычную функцию-член класса в виртуальную, достаточно в начале инструкции ее объявления поставить ключевое слово `virtual` (см. также раздел «Функ-

ции» в главе «Основные особенности программирования на C++»):

```
virtual <тип функции> <имя функции> (<список параметров>);
```

В результате такого объявления получаем виртуальную функцию, адрес которой определяется только во время выполнения программы (а не на этапе компиляции, как обычно). Определение адреса функции во время выполнения программы называется *поздним связыванием* (late binding).

Объявить функцию виртуальной достаточно один раз, в объявлении класса. Это автоматически делает виртуальными все функции производных классов, имеющие то же имя и тот же тип (включая классы, производные от его производных).

В предыдущей главе был создан класс Person (см. раздел «Наследование и оболочка объекта» в главе «Наследование»). Для простоты немного изменим типы его данных-членов и дополним функцией вывода их на экран.

Пример 44

```
#include <stdio.h>
class Person
{
    public:
        char surname[30];
        char name[30];
        char patronymic[30];
        int age;
        virtual void display(void);
//...
};
void Person::display(void)
{
    printf ("%s %s %s", surname, name, patronymic);
}
```



ПРИМЕЧАНИЕ В определение функции display включать ключевое слово virtual не нужно.

Создадим класс `EmployPerson`, производный от `Person`, добавив переменную для хранения данных о месте работы `WPlace` и определив для него новую функцию `display`. Она также будет виртуальной, несмотря на отсутствие такого определения в явном виде.

```
class EmployPerson:public Person
{
    public:
        char WPlace[80];
        void display(void);    //переопределение функции
//...
};
void EmployPerson::display(void)
{
    printf ("%s %s %s", surname, name, patronymic);
    printf ("\nМесто работы:%s", WPlace);
}
class PostEPerson:public EmployPerson
{
    public:
        char WPost[80];
        void display(void);    //переопределение функции
//...
};
void PostEPerson::display(void)
{
    printf ("%s %s %s", surname, name, patronymic);
    printf ("\nМесто работы:%s\nДолжность %s",
WPlace, WPost);
}
```

Нет никаких особых требований к работе с виртуальной функцией: она объявляется, определяется и используется точно так же, как и любая обычная функция-член класса.



ВНИМАНИЕ Виртуальные функции подчиняются всем обычным синтаксическим правилам функций-членов класса, кроме одного ограничения — такая функция не может быть встраиваемой (`inline`), а значит, ее тело не может быть определено в объявлении класса.



ПРИМЕЧАНИЕ Несмотря на простоту введения виртуальных функций, не стоит объявлять функции виртуальными, если они не будут переопределяться в производных классах.

Эффект объявления функции виртуальной особенно ярко проявляется при доступе к объектам класса через указатели базового класса (см. также раздел «Базовые классы и указатели» в главе «Наследование»).

При вызове разных функций `display` для указателей на объекты разных классов может быть использована одна и та же инструкция. В следующем примере классы `EmployPerson` и `PostEPerson` являются производными от класса `Person`.

Пример 45

```
Person *ppers1;  
//...  
ppers1 = new EmployPerson;  
ppers1.display();    //вызов EmployPerson::display  
ppers1 = new PostEPerson;  
ppers1.display();    //вызов PostEPerson::display
```

Вызов виртуальных и не виртуальных функций в C++ производится по-разному, поэтому виртуальные функции требуют немного больше места в памяти, чем не виртуальные. Однако в их синтаксисе нет различий, кроме ключевого слова `virtual` при объявлении. Эта особенность делает введение виртуальных функций в существующие проекты очень простым.

Функции, не объявленные виртуальными, также могут быть определены заново в производных классах, но при этом иногда возникают проблемы, о которых говорилось в самом конце предыдущей главы. В представленном ниже фрагменте программы последняя инструкция будет вызывать функцию `MyStr::copy`, а не `MyIOStr::copy`, как это можно было бы предположить:


```
MyIOStr IOwstr;  
MyStr *pwstr = &IOwstr;  
pwstr->copy("Привет!");
```

Такой вызов объясняется типом объекта `pwstr`, определенным при компиляции.

С другой стороны, объявив функцию `copy` виртуальной для базового класса `MyStr`, мы определяем адрес функции уже во время выполнения программы. Так как указатель `pwstr` к этому моменту указывает на объект класса `MyIOStr`, в последней инструкции вызывается функция `MyIOStr::copy`, как и ожидалось.

В следующем разделе приводится пример, более наглядно демонстрирующий преимущества виртуальных функций.

Программирование команд меню

Простым примером объектно-ориентированного подхода к разработке программ является создание меню программы. Программа выводит на экран несколько команд меню, из которых пользователь выбирает нужную, после этого выбранная команда выполняется.

Этот алгоритм довольно легко программируется с помощью процедур, но объектно-ориентированный подход имеет некоторые преимущества. Если каждая команда меню программируется как отдельный объект, то вызов функции, реализующей эту команду, выполняется самим объектом. Такое распределение позволяет избежать использования инструкции `switch` для выбора команды в главной функции `main`, а также редактирования этой инструкции при вводе в меню дополнительных команд.

Благодаря объектам программа становится более устойчивой, гибкой и легкой для чтения.

Введем простой класс `CMenuItem`, содержащий два члена: имя команды и виртуальную функцию для выполнения команды. Этот класс определяется в качестве базового для всех команд меню, и его объекты не будут создаваться в программе.

Пример 46

```
//cmenu.h – файл заголовков класса MyMenuItem
#include <stdio.h>
class CMenuItem
{
    public:
        char title[81];
        virtual void do_com(void) = 0;
};
```

Здесь нужно пояснить, что символы = 0 в конце объявления функции `do_com` определяют эту функцию как *чистую виртуальную функцию* (pure virtual function), которая отличается от остальных виртуальных функций тем, что ее определение вообще отсутствует в базовом классе и появляется только в производных классах. Таким образом, чистая виртуальная функция просто резервирует место для функций, которые будут определены позже (см. также раздел «Чистые виртуальные функции и абстрактные классы»).

Каждый объект команды меню вводит свое определение функции `do_com`. Главная программа вызывает функцию `do_com` выбранного объекта для исполнения команды.

Создадим меню для работы с файлами. Пусть оно состоит из трех команд:

- Открыть файл;
- Записать в файл строку;
- Заккрыть файл.

Каждая команда меню представляется как объект класса `CMenuItem`, дающий определение функции `do_com`. Объявление класса для каждой команды меню одинаково (за исключением имени класса) и объявляет свою функцию `do_com` для этого класса.

Файл `filecoms.cpp` состоит из трех частей: объявления всех команд меню, определения действий, выполняемых этими командами, и инициализации команд. Отметим, что файл легко редактируется при дополнении меню новой командой.

Дополнительно введем в этот файл глобальную переменную `f1` — указатель файла, с которым будут выполняться все команды, и функцию `ring` для выдачи звукового сигнала при неправильном выполнении команды.

Пример 46**(продолжение)**

```
//-----файл filecoms.cpp -----//
// объявление команд меню
#include <stdio.h>
#include <string.h>
#include "cmenu.h"
FILE *f1=NULL;
void ring (void) {putc('\007', stdout);}
class CFileOpen:public CMenuItem
{
    void do_com (void);
};
class CWriteFile:public CMenuItem
{
    void do_com (void);
};
class CFileClose:public CMenuItem
{
    void do_com (void);
};
// определение команд меню
void CFileOpen::do_com (void)
{
    char wstr[81];
    if (f1 != NULL)
    {
        printf ("\nФайл уже открыт");
        ring();
        return;
    }
    printf ("\nВведите имя файла:");
    scanf ( wstr);
    if ((f1 = fopen ( wstr,"wr")) == NULL)
```


Пример 46

(продолжение)

```
{
    printf ("\nОшибка открытия файла %s", wstr);
    ring();
}
}
void CWriteFile::do_com (void)
{
    char wstr[81];
    if (f1 == NULL)
    {
        printf ("\nНет открытого файла");
        ring();
    }
    else
    {
        printf("\nВведите строку для записи в файл:");
        scanf (wstr);
        fprintf (f1, "\n%s", wstr);
    }
}
void CFileClose::do_com (void)
{
    if (f1 == NULL)
    {
        printf ("\nНет открытого файла");
        ring();
    }
    else
    {
        fclose(f1);
        f1=NULL;
    }
}
// инициализация команд меню
int NCommands; //число команд меню
CMenuItem *coms[20]; //массив команд меню
void initcoms(void)
```



```
{
    coms[0]= new CFileOpen;
    strcpy (coms[0]->title, "Открыть файл");
    coms[1]= new CWriteFile;
    strcpy(coms[1]->title, "Записать в файл строку");
    coms[2]= new CFileClose;
    strcpy (coms[2]->title, "Закрыть файл");
    NCommands = 3;
}
```

Главная программа должна только обеспечить доступ к двум глобальным переменным: к числу команд меню `NCommands` и к массиву этих команд `coms`, объявляя их с помощью ключевого слова `extern`. Последнее необходимо, если проект программы состоит из двух или более файлов.



СОВЕТ Как уже говорилось, вариант создания проекта программы из разных файлов является более предпочтительным, так как он позволяет структурировать программу и в дальнейшем использовать отдельные файлы проекта в других программах.

Пример 46

(продолжение)

```
//-----файл main.cpp-----//
//главный файл: работа с меню
#include <stdio.h>
#include "cmenu.h"
extern int NCommands;
extern CMenuItem *coms[];
void main (void)
{
    int i,ncom;
    initcoms();
    do
    {
        puts("\nМеню:");
        for ( i=0; i < NCommands; i++)
            printf("%2d. %s\n",i+1,coms[i]->title);
        printf("%2d. Выход\nВаш выбор:", NCommands+1);
        scanf("%d", &ncom);
    }
```


Пример 46**(продолжение)**

```
        if ( ncom>0 && ncom<=NCommands)
            coms[ncom-1]->do_com();
    }
    while (ncom<= NCommands);
}
```

Очевидно, при таком подходе использование виртуальных функций необходимо. В противном случае представленная ниже инструкция всегда будет вызывать функцию `CMenuItem::do_com`:

```
coms[ncom-1]->do_com();
```

Заметим, что в данном случае были определены три команды меню для работы с файлами, но при необходимости их число можно увеличить. Более того, можно создать меню, одной из команд которого будет команда Файл, и при ее выборе будет появляться созданное нами меню. Аналогичным образом можно создать меню для редактирования текста, работы с графикой и т. д.

Достоинства виртуальных функций

Очевидно, для выполнения выбранной команды в главной программе вместо инструкции `coms[ncom-1]->do_com();` и виртуальных функций можно использовать инструкцию `switch`, как показано в примере 47.

Пример 47

```
switch(NCommands)
{
    case 1:FileOpenCommand();
           break;
    case 2:WriteToFileCommand();
           break;
    case 3:FileCloseCommand();
           break;
    default:;    //завершение программы
}
```

Однако при любом изменении структуры меню этот фрагмент программы также потребует изменений, новой ком-

пиляции и построения задачи. Кроме того, с увеличением количества команд будет увеличиваться размер главной программы и объем выполняемых ею действий, что сделает программу нечитаемой. Если главную программу и функции, реализующие меню, пишут разные программисты, задача еще более усложняется.

Для сложных и разветвленных меню особенно важно, чтобы принятие решений и распределение ресурсов, то есть максимально возможное количество действий, выполнялось в объектах меню. Такой подход позволяет освободить главную программу от деталей и избежать ее переписывания и перекомпиляции при изменении меню или функций, реализующих его команды.

Помимо этого существует еще два преимущества использования виртуальных функций.

В примере 46 мы инициализировали команды меню непосредственно в функции `initcoms`. Для этого также можно использовать и другие способы, например, чтение меню из файла данных (подобный способ реализован в компиляторе ресурсов для построения меню под Windows).

Иногда требуется изменить меню во время выполнения программы. В нашем примере команды Записать в файл строку и Закрыть файл не могут быть выполнены до использования команды Открыть файл. Открытие следующего файла возможно только после закрытия предыдущего. Недоступные команды меню могут отсутствовать или выделяться другим цветом (примеры таких меню можно найти во многих усовершенствованных программах DOS и практически в любом приложении Windows). Метод виртуальных функций позволяет просто и наглядно изменять структуру меню в любое время (для этого достаточно создать несколько вариантов инициализации команд меню).

Чистые виртуальные функции и абстрактные классы

В примере 46 при определении базового класса `CMenuItem` была введена *чистая виртуальная функция* (`pure virtual`

function) do_com. Признаком такой функции являются символы = 0 в конце ее объявления. Это означает, что определения функции нет в базовом классе, и такое определение произойдет в производных классах.

Класс, который содержит хотя бы одну чистую виртуальную функцию, называется *абстрактным классом*. Примером такого класса является класс CMenuItem. Для абстрактных классов существует важное ограничение: они не могут быть использованы напрямую для создания объектов, хотя объявление указателей на такие классы допускается:

```
CMenuItem *pMyMenu; //нормально  
CMenuItem MyMenu; //ошибка!
```

Несмотря на невозможность создания объектов абстрактного класса, они играют важную роль в программировании, так как являются базовыми классами для создания целых семейств производных классов. Хорошим примером создания и использования абстрактных классов служит класс CMenuItem.

Включение в класс чистой виртуальной функции позволяет избежать лишнего определения функции и одновременно обеспечивает присутствие ее определения в производных классах. В случае отсутствия определения такой функции в каком-либо классе компилятор выдаст ошибку при попытке создания объектов этого класса.

Определение виртуальной функции

Как правило, при программировании на любом языке приходится выбирать между размером программы и скоростью ее выполнения. При использовании виртуальных функций можно выиграть в размере программы, но при этом проиграть в скорости ее работы. Чем чаще в программе применяются виртуальные функции, особенно при их множественном наследовании, тем больше становится время работы программы.

В простейшем случае, без множественного наследования выполнение виртуальных функций относительно прос-

то и не изменяется при переходе от одной версии C++ к другой.

Рассмотрим сначала, как выполняется косвенный вызов функций через указатель (этот пример одинаково выполняется в C++ и в C).

Пример 48

```
#include <stdio.h>
float sum (float a, float b)
{return (a+b);}
float subst (float a, float b)
{return (a-b);}
void main()
{float (*pfunc)(float, float);
  float a=1.5, b=3.7;
  pfunc= &sum;
  printf("\n%7.2f+%7.2f=%7.2f", a, b, (*pfunc)(a,b));
  //сложение)
  pfunc= &subst;
  printf("\n%7.2f-%7.2f=%7.2f", a, b, (*pfunc)(a,b));
  // вычитание
}
```

Виртуальные функции дополняют такой косвенный вызов следующими особенностями:

- трактуют косвенный вызов в терминах функций-членов;
- строят таблицу указателей на функции для каждого класса;
- скрывают косвенный вызов таким образом, что синтаксически вызов виртуальной функции выглядит так же, как вызов обычной функции.

Последняя особенность приводит к тому, что при определении функции в производном классе можно даже забыть о том, что она виртуальная. Простота синтаксиса виртуальных функций при тех преимуществах, которые они предоставляют программисту, делает их использование еще более привлекательным.

Механизм вызова виртуальных функций



СОВЕТ Читатели, только начинающие знакомство с C++, могут обратиться к этому разделу позже.

Создадим базовый класс `CMyBaseClass`, членами которого являются три виртуальные функции.

Пример 49

```
class CMyBaseClass
{
    public:
    virtual void function1(int i, float f);
    virtual int function2(void);
    virtual double function3(double pArray[]);
};
```

Компилятор C++ строит таблицу `VTable` указателей на определение каждой из виртуальных функций класса `CMyBaseClass`. Следует отметить, что для всех объектов класса строится одна таблица, так как для всех объектов одного класса создается только по одному экземпляру функций-членов этого класса (см. также раздел «Объекты класса» в главе «Классы»). Указатели на функции, не являющиеся виртуальными, не входят в эту таблицу, так как вызов этих функций выполняется обычным способом.

Для каждого производного класса создается своя таблица, начало которой составляют указатели на определения виртуальных функций базового класса, заново переписанные в данном производном классе. Если производный класс имеет свои виртуальные функции, то указатели на их определения добавляются в конец таблицы.

Каждый объект какого-либо класса имеет скрытый член, который располагается в начале объекта и указывает на таблицу `VTable`, определенную для этого класса. Назовем его `pVTable`. (Если в определении класса отсутствуют виртуальные функции, таблица `VTable` для него не создается,

и член `pVTable` отсутствует в объекте.) В действительности объект «не знает» тип этого указателя: он используется только для косвенного вызова функций.

Рассмотрим вызов одной из виртуальных функций, объявленных в примере 49.

```
CMyBaseClass myobj1, *pmyobj;  
pmyobj = & myobj1;  
pmyobj -> function2();
```

Компилятор C++ транслирует вызов функции из последней строки в следующий косвенный вызов функции:

```
(* (pmyobj->pVTable-> function2))();
```

 **ПРИМЕЧАНИЕ** В языке C отсутствует понятие виртуальных функций, но их можно симулировать. В этом случае для каждого «класса» должна быть создана и инициализирована структура, реализующая таблицу `VTable`, для каждого «объекта» создан и инициализирован указатель `pVTable`. Далее реализуется косвенный вызов функции в соответствии с указанным выше алгоритмом.

Часть III

Приложения

Отличия С и С++

Вопреки мнению, что для перехода от С к С++ достаточно простой перекомпиляции программы, между этими языками существует довольно много различий.

- Функции С++, тип возвращаемого результата которых отличен от `void`, должны возвращать значение соответствующего типа с помощью инструкции `return`. В С при отсутствии такой инструкции может выводиться предупреждение, в С++ — это ошибка.
- Заголовки всех функций, используемых в программе С++, должны присутствовать в виде прототипов, то есть с указанием всех параметров и их типов, а не в виде кратких заголовков, применявшихся в ранних версиях С.
- Язык С++ выполняет более строгую проверку типов, в частности нельзя выполнять присваивание указателей на разные типы или указателя на тип `void` без приведения их типов к одному. В С при отсутствии оператора приведения типа в таких случаях выводится предупреждение, в С++ — это ошибка. Использование одного из операторов приведения типов позволяет решить проблему.
- С++ требует обязательного и полного объявления функции и ее параметров перед ее использованием. В С при отсутствии явного объявления подразумевается, что переменные имеют значения типа `int`, и, хотя такой стиль программирования не рекомендуется, никаких диагностических сообщений при этом не выводится. По той же причине главная функция `main()` в С++ всегда должна быть объявлена как `int main()` или `void main()`.
- В связи с тем, что идентификаторы глобальных переменных, имена новых типов, объявленных программистом, имена структур и объединений, а также имена

функций находятся в С++ в одном пространстве имен, недопустимо использование одного и того же имени для этих идентификаторов. В С можно было назвать одним и тем же именем глобальную переменную или имя функции и имя, определенное с помощью оператора определения типа `typedef`, в С++ это вызовет ошибку.

Существует еще одна особенность, которую следует учитывать, если используются модули программ, написанных на С и С++. В С++ имеется возможность перегрузки функций в зависимости от количества и типа параметров. Однако чтобы программа идентифицировала эти функции, их имена должны отличаться. Поэтому С++ добавляет к имени функции, указанному в программе, специальные символы и записывает их вместе с именами в объектные файлы. Кроме того, переменные разного типа с одним и тем же именем при построении (линковке) программы из разных файлов также рассматриваются как разные переменные и хранятся как имена переменных со специальными символами, обозначающими их тип. В языке С такой необходимости расширения идентификаторов нет.

В результате объектные файлы модулей С и С++ не смогут распознавать символы друг друга. К счастью, С++ обеспечивает хороший выход из этого положения: нужно просто использовать объявления `extern "C"` в модулях С++, чтобы ввести символы, определенные в С. Объявление внешней переменной этим способом приведет к тому, что присоединение специальных символов, обычно выполняемое в С++, будет подавлено, и переменная получит то же имя, что в С. Например:

```
extern "C"
{
    int mode; //mode как переменная С
}
```

Переход от С++ к С выполнить сложнее, так как в С++ поддерживается значительно больше возможностей, чем в С. Необходимость такого перехода маловероятна, но если это потребовалось, скорее всего придется переписать зна-

чительную часть программы. Несмотря на то что С++ строже относится к использованию различных типов, он обеспечивает большую, по сравнению с С, свободу.

- В С++ можно размещать объявления и другие инструкции в любом порядке. В С нужно располагать все объявления локальных переменных в начале функции.
- В отличие от С, в котором при инициализации переменных можно использовать только константы, в С++ для инициализации переменной или объекта можно указывать любое верное выражение.
- При объявлении класса, в том числе структуры и объединения, его имя автоматически становится именем типа, которое можно использовать наряду со всеми остальными типами. В С определение `struct MyClass` нужно для объявлений и оператора приведения типа; полноценное имя типа создается с помощью оператора `typedef`. Для совместимости с С в С++ поддерживается синтаксис, подобный `struct MyClass` и `class MyClass`, но такая запись не является необходимой.
- Возможность определять оставшуюся часть строки как комментарий с помощью двойного слэша (`//`) отсутствует в некоторых компиляторах С, для обозначения комментария в них допустимы только символы `/*...*/`.
- Ввод и вывод в С можно выполнять только с помощью функций `printf`, `scanf` и им подобных, определенных в файле заголовков `stdio.h`. Объектов ввода/вывода `cin` и `cout` в С нет.

При создании программ, которые должны компилироваться как в С, так и в С++, например, библиотечных функций для разных платформ, следует внимательно отслеживать ограничения как для С, так и для С++.

Особенности ANSI C++

После многих лет усовершенствований стандарт ANSI C++ сейчас относительно стабилен. Это означает, что его общепринятая версия в ближайшее время, вероятно, не потребует значительных изменений.

Многие из новых особенностей ANSI C++ являются расширениями самого языка (например, ключевые слова `mutable` и `explicit`), при этом их использование не обязательно. Чаще всего определение ANSI вводит новую методику программирования, одновременно поддерживая и старые методы. Однако рано или поздно придет время, когда поддержка всех старых классов и функций прекратится и программист, не освоивший новую методику, останется не у дел. Хорошим способом предотвратить такую ситуацию является изучение новых возможностей и подходов.

К основным особенностям ANSI C++ относятся следующие:

- новый стиль файлов заголовков;
- операторы приведения типов ANSI;
- шаблоны и обработка исключений;
- новые ключевые слова;
- область видимости в условной инструкции `if`;
- перегрузка функций с типом `enum`;
- ссылки вперед для вложенных классов.

Первые два раздела предлагают новые, более рациональные и предпочтительные способы выполнения обычных действий. Остальные разделы описывают новые возможности языка.

Новый стиль файлов заголовков

Традиционная директива `#include` служит для чтения файла заголовков и включения его содержимого в другой файл.

Файл заголовков всегда находится на физическом диске и может быть прочитан так же, как и всякий другой.

Новый подход заключается в подключении виртуальных файлов заголовков для стандартной библиотеки. Прототипы функций стандартных библиотек в таком случае определяются через пространство имен, к которому они относятся (например, `cstdlib`, `cstdio`, `cmath`). Компилятор при этом может читать соответствующий файл с диска или загружать определения функций каким-то другим способом. Это изменение было введено, главным образом, для увеличения эффективности C++ в будущем.

Таким образом, вместо директивы

```
#include <stdio.h>
```

в ANSI C++ предлагается использовать следующую директиву:

```
#include <cstdio>
```

В описании функций стандартной библиотеки указаны имена как обычных, так и виртуальных файлов заголовков. В дальнейшем предполагается полный переход к виртуальным файлам заголовков.

Операторы приведения типов ANSI

С и ранние версии C++ поддерживали один оператор приведения типа для всех случаев. Сейчас этот оператор также можно применять, но требования к соответствию типов ужесточились, особенно это касается приведения типов указателей.

Рассмотрим оператор приведения типа к целому (`int*`), синтаксис которого был определен еще в C:

```
int pint = (int*) malloc (n);
```

Этот оператор по-прежнему является правильным, но вместо него ANSI C++ предлагает использовать один из четырех новых операторов: `const_cast`, `dynamic_cast`, `reinterpret_cast` и `static_cast`.

Только один из них, `dynamic_cast`, предоставляет совершенно новую возможность. Если поддержка информации времени выполнения программы (RTTI) включена (см. документацию компилятора), он помогает определить точный тип объекта, находящегося по адресу, определенному указателем. Использование оператора `dynamic_cast` связано с оператором `typeid`, также являющегося новой особенностью ANSI C++ и требующего включения поддержки RTTI.

Оставшиеся три оператора выполняют такое же приведение типа, как и оператор старого стиля, но цели каждого из них более конкретны. В частности, `const_cast` и `reinterpret_cast` являются потенциально опасными операторами приведения типов, и инструкции, в которых они находятся, требуют особого внимания.

Например, первый пример оператора можно записать таким образом:

```
int pint=reinterpret_cast(int*)(malloc(n));
```

(См. также главу «Операторы приведения типов».)

Шаблоны и обработка исключений

Шаблоны и обработка исключений — две главные отличительные черты C++, не представленные в ранних версиях. Шаблоны позволяют создавать общий класс или функцию, в которые затем можно подставлять конкретный тип. Обработка исключений — отличный метод для обработки ошибок во время выполнения программы и других событий. Ее использование гораздо более удобно, чем старых функций C `raise` и `signal`, обеспечивавших обработку только небольшого количества событий.

Спецификация ANSI сейчас официально требует полной поддержки как шаблонов, так и обработки исключений. Подробнее о них рассказывается в главе «Ключевые слова» (разделы «`template`», «`try`», «`throw`», «`catch`»).

Кроме ключевого слова `template` для поддержки шаблонов ANSI C++ предлагает большой набор классов из стандартной библиотеки шаблонов (Standard Template Library).

Новые ключевые слова

Помимо перечисленных выше спецификация ANSI поддерживает следующие ключевые слова:

- `mutable` — модификатор объявления членов, разрешающий их изменение даже в случае объявления их с модификатором `const`;
- `explicit` — отказ от преобразования конструктора;
- `bool` — тип данных, имеющих только два значения: `true` и `false`;
- `wchar_t` — широкий символьный формат данных, в котором каждый символ занимает два байта, его применение делает возможным использование не европейских алфавитов.

Подробнее о ключевых словах `mutable` и `explicit` рассказывается в соответствующих разделах главы «Ключевые слова», а о типах `bool` и `wchar_t` — в главе «Типы данных».

Область видимости в условной инструкции `if`

Согласно ANSI C++ можно объявлять переменную внутри условия инструкции `if`. Объявленная таким образом переменная видна только внутри этой инструкции. Например:

```
if (short k=0)
{
    cout << "New number:";
    cin >> newk;
}
k = newk; //Ошибка: k не определено!
```

Как видно из примера, переменная объявляется не в блоке инструкций, а в самом условии условной инструкции. Заметим, что знак равенства (=) в примере означает присваивание значения 1 переменной `k`, а не проверку на равенство `k` и 1.

Наиболее полезным применением этого свойства является возможность управлять областью видимости указателей, возвращаемых с помощью оператора `dynamic_cast`.


```
if(A *pA=dynamic_cast<param>)  
{  
    //выполняется только, если param  
    // является указателем на объект класса A  
    pA->f1(); //вызов функции-члена класса A  
    //...  
}  
pA->f1(); //Ошибка: pA не определен!
```

Здесь указатель pA можно использовать только внутри условной инструкции. Главное преимущество такой формы объявления переменной состоит в том, что компилятор автоматически предотвратит неверное использование указателя pA там, где он не определен (см. также главу «Операторы приведения типов»).

Перегрузка функций с перечисляемыми типами

Спецификация ANSI расширяет возможности работы с перечисляемыми типами, то есть при перегрузке функций перечисляемый тип и другие целые типы различаются. Рассмотрим пример:

```
enum Seasons  
{  
    winter, spring, summer, autumn  
};  
int f1 (int k)  
{  
    cout<< "f1(int)"<<endl;  
}  
int f1 (Seasons k)  
{  
    cout<< "f1(Seasons)" <<endl;  
}
```

В этом примере при вызове функции

```
f1(spring);
```

будет выведена строка:

```
f1(Seasons)
```


Ссылки вперед для вложенных классов

Возможность ссылок на классы (или структуры в С) без их предварительного определения (ссылки вперед) существовала еще в С (см. также раздел «Ссылка вперед» в главе «Словарь терминов и понятий»). Однако в ANSI C++ эта возможность расширена для вложенных классов.

```
class A
{
    class B; //ссылка на класс B
    B *pB;   //ссылка вперед
    class B
    {
        //вложенное определение класса B
        //...
    };
};
```

Можно определить вложенный класс и вне содержащего его класса с использованием оператора определения области видимости (::):

```
class A
{
    class B; //ссылка на класс B
    B *pB;   //ссылка вперед
};
class A::B
{
    //вложенное определение класса B
    //...
};
```


Стандартные исключения

Как уже отмечалось в главе «Библиотечные функции», в С++ можно вырабатывать и перехватывать исключения любого типа. В том числе, можно перехватывать стандартные исключения, определенные в самом языке С++ и возникающие автоматически, как реакция на определенную ситуацию. Для появления таких исключений использовать ключевое слово `throw` не нужно.

Базовый класс стандартных исключений С++ называется `exception`. Он объявляет конструктор и деструктор, но не имеет членов (точное объявление исключения можно найти в документации компилятора).

Класс исключений делится на два подкласса.

- Исключения класса `logic_error` сообщают о действиях, которые не могут быть выполнены. Условия, определяемые во время выполнения программы, делают дальнейшее выполнение каких-либо действий бессмысленным.
- Исключения класса `runtime_error` сообщают о неправильных результатах действий. Такие исключения возникают в результате действий, начальные условия выполнения которых были верными, например, параметры функции были правильными, но окончательный результат получился неверным.

Вообще говоря, все стандартные исключения возникают как ошибки выполнения программы. Однако исключения `logic_error` являются результатом попытки выполнить что-либо, что не имеет смысла уже до этой попытки, например, вызов функции с неправильными параметрами выполнения. Категория `runtime_error` включает действия, ошибочность которых проявляется в процессе или после выполнения.

В следующем списке приведены специфические классы исключений общего типа (в скобках указан их базовый класс):

- `domain_error (logic_error)` — нарушение внутренних условий перед выполнением действия;
- `invalid_argument (logic_error)` — попытка вызова функции с неверным параметром;
- `length_error (logic_error)` — попытка создания объекта большего, чем максимальный размер типа `size_t`;
- `out_of_range (logic_error)` — попытка вызова функции с параметром, не входящим в допустимые значения;
- `bad_cast (logic_error)` — неправильное использование оператора `dynamic_cast` (см. также главу «Операторы приведения типов»);
- `bad_typeid (logic_error)` — использование неправильно-го значения, определяющего тип (`type_info`);
- `range_error (runtime_error)` — неправильный результат выполнения;
- `overflow_error (runtime_error)` — арифметическое переполнение (результат слишком большой для указанного типа данных);
- `bad_alloc (runtime_error)` — ошибка при динамическом распределении памяти.

Таблица ASCII-кодов

Код	Символ	Код	Символ
00	NUL	29	Разделитель группы
01		30	Разделитель записи
02		31	US
03		32	Пробел
04		33	!
05		34	"
06	АСК	35	#
07	Звонок	36	\$
08	Удаление назад	37	%
09	Табуляция	38	&
10	Перевод строки	39	,
11		40	(
12	FF	41	)
13	Возврат каретки	42	*
14		43	+
15		44	-
16		45	
17		46	'
18		47	/
19		48	0
20		49	1
21	NAK	50	2
22	SYN	51	3
23		52	4
24		53	5
25		54	6
26		55	7
27	Esc	56	8
28	Разделитель файлов	57	9

продолжение ➤

Код	Символ	Код	Символ
58	:	93	]
59	;	94	^
60	<	95	_
61	=	96	`
62	>	97	a
63	?	98	b
64	@	99	c
65	A	100	d
66	B	101	e
67	C	102	f
68	D	103	g
69	E	104	h
70	F	105	i
71	G	106	j
72	H	107	k
73	I	108	l
74	J	109	m
75	K	110	n
76	L	111	o
77	M	112	p
78	N	113	q
79	O	114	r
80	P	115	s
81	Q	116	t
82	R	117	u
83	S	118	v
84	T	119	w
85	U	120	x
86	V	121	y
87	W	122	z
88	X	123	{
89	Y	124	/
90	Z	125	}
91	[	126	~
92	\	127	Удаление

Список характерных задач в C++

В представленной ниже таблице содержится список часто выполняемых при разработке программ действий, а также стандартные элементы C++, позволяющие их реализовать. Для простоты поиска необходимых элементов в таблице используются ключевые понятия, которые значительно его упрощают. Например действия, которые нужно выполнить для аварийного завершения программы, можно искать по словам *аварийный*, *выход* и *программа*.



ВНИМАНИЕ Ключевые понятия, используемые для описания выполняемых действий, не относятся к стандартным ключевым словам языка C++.

Выполняемое действие	Элемент C++
Вычисление <i>абсолютного</i> значения	abs, labs, fabs
<i>Аварийный</i> выход из программы	abort
Получение <i>адреса</i> значения	&
Установка <i>адреса</i> для длинного перехода	setjmp
Определение <i>алгоритма</i> быстрой сортировки	qsort
Вычисление <i>арксинуса</i>	asin
Вычисление <i>арктангенса</i>	atan, atan2
Вычисление <i>арккосинуса</i>	acos
Вставка фрагмента программы на <i>ассемблере</i>	asm
Объявление виртуальной функции или базового класса	virtual
<i>Безусловный</i> переход	goto
Установка <i>битов</i>	bitwise OR ()
Операции над <i>битами</i> (И, ИЛИ, НЕ, исключающее ИЛИ)	&, , ~, ^
Инверсия <i>битов</i>	~

продолжение ➤

Выполняемое действие	Элемент C++
Сброс <i>битов</i>	bitwise AND (&)
Выполнение операций над <i>блоками памяти</i>	memchar, memcmp, memcpy, memmove, memset
Сравнение двух <i>блоков памяти</i>	memcmp
Копирование <i>блока памяти</i>	memcpy
Уменьшение размера <i>блока памяти</i>	realloc
<i>Больше или равно</i>	>=
<i>Больше чем</i>	>
<i>Булев тип</i>	bool
Установка <i>буфера</i> для файловых операций	setbuf, setvbuf
Очистка <i>буфера</i> файла	fflush
Очистка <i>буфера</i> ввода/вывода	fflush
Отказ от <i>буферизованной</i> работы с файлами	setbuf, setvbuf
Выбор <i>варианта</i>	switch
Объявление типа записи с <i>вариантами</i>	union
<i>Ввод</i> данных из потока ввода	cin
<i>Ввод</i> (чтение) данных с консоли	scanf
<i>Ввод</i> (чтение) двоичных данных из файла	fread
<i>Ввод</i> (чтение) очередного символа из файла	fgetc, getc
<i>Ввод</i> (чтение) очередного символа с консоли	getchar
Использование строки как объекта потока <i>ввода</i>	istrstream
<i>Ввод</i> (чтение) строки с консоли	gets
Объекты <i>ввода/вывода</i> для потоков	istream, ostream
Возврат символа в поток <i>ввода</i>	ungetc
<i>Ввод</i> объекта из потока	istream
<i>Ввод</i> значений из строки	sscanf
Чтение строки <i>ввода</i>	gets
Форматированный <i>ввод</i> в файл	fscanf
Поддержка множественности <i>версий</i> программы	#if
Проверка символов на принадлежность к символам <i>верхнего регистра</i>	isupper

Выполняемое действие	Элемент C++
Определение области <i>видимости</i> для класса	::
Объявление <i>виртуальной</i> функции или базового класса	virtual
Включение файла заголовков	#include
Объявление <i>внешней</i> переменной	extern
Определение <i>внутреннего</i> доступа	private
Разрешение доступа к <i>внутренним</i> членам	friend
<i>Возврат</i> символа в поток ввода	ungetc
<i>Возврат</i> в вызвавшую функцию	return
<i>Возврат</i> текущего объекта	this
Установка <i>возвращаемого значения</i>	return
<i>Возведение</i> числа «е» в степень	exp
<i>Возведение</i> любого числа в степень	pow
<i>Возбуждение</i> исключения	throw
Время компиляции	__TIME__, __DATE__
Вывод <i>времени</i> в формате	strftime
Преобразование <i>времени</i> по Гринвичу	gmtime
Получение <i>времени</i> выполнения программы	clock
Получение системного <i>времени</i> и даты	time
Создание <i>временного</i> файла	tmpfile
<i>Вставка</i> фрагмента программы на ассемблере	asm
<i>Выбор</i> варианта	switch
Вывод <i>времени</i> в формате	strftime
Вывод дня недели	strftime
Вывод данных в строку с форматированием	sprintf
Вывод двоичных данных в файл	fwrite
Вывод данных на консоль	putchar, puts, printf
Вывод символов в файл	fputc, putc
Объект <i>вывода</i> для потока	ostream
Вывод в поток ошибок	cerr
Вывод строки в файл	fputs
Форматированный <i>вывод</i> в файл	fprint

Выполняемое действие	Элемент C++
Объекты потоков <i>ввода/вывода</i>	<code>fstream, ifstream, ofstream</code>
Использование строки как объекта потока <i>вывода</i>	<code>ostrstream</code>
<i>Вывод</i> данных на экран	<code>putchar, puts, printf</code>
<i>Вывод</i> значений переменных	<code>printf</code>
<i>Вывод</i> очередного символа в файл	<code>fputc, putc, putchar</code>
<i>Вывод</i> ошибки компиляции	<code>#error</code>
<i>Вывод</i> названия месяца	<code>strftime</code>
<i>Вывод</i> в поток	<code>cout</code>
<i>Выделение</i> памяти для массива объектов	<code>new []</code>
Увеличение размера <i>выделенного</i> блока памяти	<code>realloc</code>
<i>Выделение</i> памяти для объекта	<code>new</code>
<i>Выделение</i> памяти в байтах	<code>malloc</code>
<i>Выделение</i> блока памяти заново	<code>realloc</code>
<i>Выделение</i> памяти для массива элементов	<code>calloc</code>
Освобождение <i>выделенной</i> памяти	<code>free, delete</code>
<i>Выделение</i> памяти для регистра	<code>register</code>
Получение типа во время <i>выполнения</i> программы	<code>dynamic_cast, typeid</code>
Переход к <i>выполнению</i> следующей итерации цикла	<code>continue</code>
<i>Выполнение</i> цикла по счетчику	<code>for</code>
<i>Выполнение</i> до тех пор, пока условие истинно	<code>while</code>
Повторение <i>выполнения</i>	<code>do while</code>
<i>Выполнение</i> цикла определенное количество раз	<code>for</code>
<i>Выполнение</i> форматированного ввода из файла	<code>fscanf</code>
<i>Выполнение</i> аварийного прерывания программы	<code>abort</code>
<i>Выполнение</i> нормального выхода из программы	<code>exit</code>
<i>Выполнение</i> очистки буфера файла	<code>fflush</code>

Выполняемое действие	Элемент C++
Выполнение удаления файла с диска	remove
Выполнение команды на уровне системы	system
Выход из цикла	break
Выход из функции	return
Вычисление абсолютного значения	abs, labs, fabs
Вычисление арккосинуса	acos
Вычисление арксинуса	asin
Вычисление арктангенса	atan, atan2
Вычисление гиперболического косинуса	cosh
Вычисление гиперболического синуса	sinh
Вычисление гиперболического тангенса	tanh
Вычисление десятичного логарифма	log10
Вычисление дробной части числа	modf
Вычисление квадратного корня	sqrt
Вычисление косинуса	cos
Вычисление мантиссы и экспоненты	frexp
Вычисление натурального логарифма	log
Вычисление остатка от деления	%, fmod, div, ldiv
Вычисление степени	pow
Вычисление степени «e»	exp
Вычисление тангенса	tan
Вычисление целой части числа	modf
Вычитание	-
Присваивание с вычитанием	--
Декремент (вычитание 1 из значения)	--
Вычисление гиперболического косинуса	cosh
Вычисление гиперболического синуса	sinh
Вычисление гиперболического тангенса	tanh
Ввод данных из потока ввода	cin
Получение данных по указателю	*
Чтение данных с клавиатуры	scanf
Получение даты	time

Выполняемое действие	Элемент C++
Ввод (чтение) <i>двоичных</i> данных из файла	fread
Вывод <i>двоичных</i> данных в файл	fwrite
Выполнение <i>двоичного</i> поиска	bsearch
Запись <i>двоичных</i> данных в файл	fwrite
Чтение <i>двоичных</i> данных из файла	fread
Тип <i>двухбайтовое</i> (короткое) целое	short
<i>Декремент</i> (вычитание 1 из значения)	--
Вычисление остатка от деления	%, fmod, div, ldiv
<i>Деление</i> по модулю	%, fmod
<i>Деление</i> с присваиванием	/=
<i>Деление</i> чисел	/, div, ldiv
Вычисление <i>десятичного</i> логарифма	log10
<i>Динамическое</i> выделение памяти	malloc, new
<i>Динамическое</i> создание объекта	new
Сброс буфера файла на <i>диск</i>	fflush
Удаление файла с <i>диска</i>	remove
Получение <i>длины</i> строки	strlen
Тип <i>длинное</i> целое	long
Вывод <i>дня недели</i>	strftime
Получение <i>дня недели</i>	mktime, time
<i>Доступ</i> к изменяющемуся объекту данных	volatile
<i>Доступ</i> к членам объекта	.
Запрет <i>доступа</i> к члену класса извне	private
<i>Доступ</i> к члену через указатель	->
Ограничение <i>доступа</i> к членам	private, protected
Определение <i>внутреннего доступа</i>	private
Определение <i>защищенного доступа</i>	protected
Определение <i>общего доступа</i>	public
Разрешение <i>доступа</i> к внутренним членам	friend
Разрешение <i>доступа</i> к члену класса извне	public
Расширение <i>доступа</i> к производному классу	protected
Вычисление <i>дробной</i> части числа	modf

Выполняемое действие	Элемент C++
Удаление <i>дробной</i> части из числа	modf
Регистрация (определение) функции завершения программы	atexit
Включение файла <i>заголовков</i>	#include
Чтение файла <i>заголовков</i>	#include
<i>Запись</i> двоичных данных в файл	fwrite
<i>Запись</i> (вывод) очередного символа в файл	fputc, putc, putchar
Объявление типа <i>записи</i> с вариантами	union
<i>Запрет</i> доступа к члену класса извне	private
<i>Запрет</i> изменения переменной	const
<i>Запрет</i> преобразования	explicit
<i>Запрос</i> блока памяти	malloc, new
Определение <i>защищенного</i> доступа	protected
Ввод <i>значений</i> с клавиатуры	scanf
Вывод <i>значений</i> переменных	printf
Ввод <i>значений</i> из строки	sscanf
Вычисление абсолютного <i>значения</i>	abs, labs, fabs
Получение <i>значения</i> pi	atan
Формирование возвращаемого <i>значения</i>	return
Логическое И	&&
Поразрядное (побитовое) И	&
Разрешение использования <i>идентификаторов</i> из (пространства имен)	using
<i>Изменение</i> имени файла	rename
<i>Изменение</i> типа указателя	reinterpret_cast
<i>Запрет</i> на <i>изменение</i> переменной	const
Разрешение <i>изменения</i> члена константы	mutable
Объявление нового <i>имени</i> типа	typedef
Определение класса или (области видимости) <i>имени</i>	::
<i>Инверсия</i> битов	~
<i>Инверсия</i> логического значения	!
<i>Инкремент</i> (сложение значения с 1)	++
Возбуждение <i>исключения</i>	throw

Выполняемое действие	Элемент C++
Начало блока обработки <i>исключений</i>	try
Обработка <i>исключения</i>	catch
Создание <i>исключения</i>	throw
хор (<i>исключающее ИЛИ</i>)	~
<i>Использование</i> не изменяющейся переменной	const
<i>Использование</i> символов из пространства имен	using
<i>Использование</i> логики переключателя	switch
<i>Использование</i> потока файла ввода	ifstream
<i>Использование</i> списка параметров переменной длины	va_arg, va_end, va_start
Разрешение <i>использования</i> идентификаторов из пространства имен	using
Разрешение <i>использования</i> пространства имен	using
Вычисление <i>квадратного</i> корня	sqrt
Ввод значений с <i>клавиатуры</i>	scanf
Чтение символа с <i>клавиатуры</i>	getchar
Ввод очередного символа с <i>клавиатуры</i>	getchar
Чтение строки с <i>клавиатуры</i>	gets
Объявление <i>класса</i>	class, struct, union
Запрет доступа к члену <i>класса</i> извне	private
Определение <i>класса</i> или области видимости имени	::
Определение шаблона <i>класса</i> или функции	template
Разрешение доступа к члену <i>класса</i> извне	public
Расширение доступа к производному <i>классу</i>	protected
Ввод <i>командной</i> строки	main
Выполнение <i>команды</i> на уровне системы	system
<i>Комбинация</i> бит	bitwise OR ()
<i>Комбинация</i> строк (конкатенация)	strcat
Получение времени <i>компиляции</i>	__TIME__

Выполняемое действие	Элемент C++
Сообщение времени <i>компиляции</i>	__TIME__, __DATE__
Сообщение об ошибке во время <i>компиляции</i>	#error
Получение параметров <i>командной строки</i>	main
Условная <i>компиляция</i>	#if
<i>Компиляция</i> функции как встроенной	inline
Предотвращение <i>конфликта</i> имен	namespace
<i>Конкатенация</i> строк	strcat
Ввод (чтение) данных с <i>консоли</i>	scanf
Ввод (чтение) очередного символа с <i>консоли</i>	getchar
Ввод (чтение) строки с <i>консоли</i>	gets
Вывод данных на <i>консоль</i>	putchar, puts, printf
Разрешение изменения члена <i>константы</i>	mutable
Разрешение изменений в <i>константе</i>	mutable
Определение символической <i>константы</i>	#define, enum
Определение списка <i>констант</i>	enum
Определение достижения <i>конца</i> файла	feof
Определение условия <i>конца</i> файла	feof
<i>Копирование</i> блока памяти	memcpy
<i>Копирование</i> массива	memcpy
<i>Копирование</i> строки	strcpy
Тип двухбайтовое <i>короткое</i> целое	short
Вычисление квадратного <i>корня</i>	sqrt
Вычисление <i>косинуса</i>	cos
Вычисление гиперболического <i>косинуса</i>	cosh
Вычисление функции, обратной <i>косинусу</i>	acos
<i>Логическое</i> отрицание	!
Инверсия <i>логического</i> значения	!
Вычисление десятичного <i>логарифма</i>	log10
Вычисление натурального <i>логарифма</i>	log
<i>Логическое</i> И	&&
Вычисление <i>мантиссы</i> и экспоненты	frexp

Выполняемое действие	Элемент C++
Выделение памяти для массива объектов	new []
Выделение памяти для массива элементов	calloc
Копирование массива	memcpy
Обращение к элементу массива	[]
Поиск значения в массиве	bsearch
Поиск в большом массиве	bsearch
Размер массива	sizeof
Сортировка массива	qsort
Сравнение массивов	memcmp
Метка оператора case	case
Метка перехода по switch	case
Меньше чем	<
Меньше или равно	<=
Вывод названия месяца	strftime
Деление по модулю	%, fmod
Вывод данных на монитор	putchar, puts, printf
Вычисление натурального логарифма	log
Использование не изменяющейся переменной	const
Проверка неравенства	!=
Проверка символов на принадлежность к символам нижнего регистра	islower
Преобразование символов к символам нижнего регистра	tolower
Получение номера текущей строки исходного файла	__LINE__
Обработка исключения	catch
Установка обработчика сигнала	signal
Определение общего доступа	public
Объединение строк	strcat
Объявление типа объединение	union
Объект потока вывода	ostream
Объект потока ввода	ifstream
Ввод объекта из потока	istream
Выделение памяти для объекта	new
Возврат текущего объекта	this

Выполняемое действие	Элемент C++
Выделение памяти для массива объектов	new []
Доступ к членам объекта	.
Доступ к членам объекта через указатель	->
Доступ к изменяющемуся объекту данных	volatile
Использование строки как объекта потока ввода	istream
Использование строки как объекта потока вывода	ostream
Освобождение объекта из памяти	delete
Получение указателя на текущий объект	this
Создание одного или более объектов	new
Удаление объекта	delete
Объявление виртуальной функции (функции позднего связывания) или базового класса	virtual
Объявление внешней переменной	extern
Объявление класса	class, struct, union
Объявление нового типа	class, struct, union
Объявление (создание) нового типа	typedef
Объявление переменной типа ссылки	&
Объявление перечисляемого типа	enum
Объявление полиморфной функции	virtual
Объявление пространства имен	namespace
Объявление разделяемых переменных	extern
Объявление типа записи с вариантами	union
Объявление типа, вводимого программистом	class, struct, union, typedef
Объявление типа объединения	union
Объявление указателя	*
Определение и объявление функции оператора	operator
Объявление шаблона класса или функции	template
Ограничение доступа к членам	private, protected

продолжение ➤

Выполняемое действие	Элемент C++
Округление с избытком	ceil
Округление с недостатком	floor
Определение и объявление функции оператора	operator
Перегружаемый оператор	operator
Операторы работы с потоками	<<, >>
Расширенные операторы работы с потоками	istream, ostream
Логические операторы (И, ИЛИ, НЕ)	&&, , !
Операторы для операций над битами (И, ИЛИ, НЕ, исключающее ИЛИ)	&, , ~, ^
Выполнение операций над строками	str<op>
Установка буфера для операций над файлами (для потока)	setbuf, setvbuf
Поразрядные операторы (И, ИЛИ, НЕ, исключающее ИЛИ)	&, , ~, ^
Определение алгоритма быстрой сортировки	qsort
Определение внутреннего доступа	private
Определение защищенного доступа	protected
Определение использования C++	__cplusplus
Определение класса или (области видимости) имени	::
Определение макрофункции (макроса)	#define
Определение области видимости класса	::
Определение общего доступа	public
Определение и объявление функции оператора	operator
Определение регистровой переменной	register
Отказ от определения символа	#undef
Удаление определения символа	#undef
Определение символической константы	#define, enum
Определение синонима для типа	typedef
Определение списка констант	enum
Определение точки входа в программу	main
Регистрация определения функции завершения программы	atexit

Выполняемое действие	Элемент C++
Определение шаблона класса или функции	template
Определение символа	#define
Освобождение выделенной памяти	free, delete
Освобождение объекта	delete
Освобождение объекта из памяти	delete
Вычисление остатка от деления	%, fmod, div, ldiv
Отказ от буферизованной работы с файлами	setbuf, setvbuf
Отказ от определения символа	#undef
Открытие файла	fopen.
Открытие файла заново	freopen
Логическое отрицание	!
Поразрядное отрицание	~
Очистка буфера ввода/вывода	fflush
Очистка буфера файла	fflush
Очистка флага ошибок	clearerr
Вывод ошибки компиляции	#error
Вывод в поток ошибок	cerr
Сообщение об ошибке во время компиляции	#error
Сброс (очистка) флага ошибок	clearerr
Получение ошибки файла	ferror
Выделение памяти в байтах	malloc
Выделение блока памяти заново	realloc
Выделение памяти для массива объектов	new []
Выделение памяти для массива элементов	calloc
Выделение памяти для объекта	new
Выделение памяти для регистра	register
Освобождение выделенной памяти	free, delete
Требование прямого доступа к памяти	volatile
Получение индекса символа в блоке памяти	memchr
Сдвиг значений внутри блока памяти	memmove
Установка значений в блоке памяти	memset

продолжение ➤

Выполняемое действие	Элемент C++
Увеличение размера выделенного блока <i>памяти</i>	realloc
Использование списка <i>параметров</i> переменной длины	va_arg, va_end, va_start
Получение <i>параметров</i> (командной строки)	main
Получение установленных <i>параметров</i> среды	getenv
Использование типа как <i>параметра</i>	template
<i>Переименование</i> файла	rename
Использование логики <i>переключателя</i>	switch
<i>Переход</i> безусловный	goto
<i>Переход</i> в зависимости от проверяемого значения	switch
Объявление внешней <i>переменной</i>	extern
Объявление <i>перечисляемого</i> типа	enum
<i>Перегружаемый</i> оператор	operator
Расположение <i>переменной</i> в регистре процессора	register
<i>Переход</i> в зависимости от значения	switch
Установка адреса для длинного <i>перехода</i>	setjmp
<i>Переход</i> между функциями	longjmp
Метка <i>перехода</i> по switch	case
<i>Переход</i> к выполнению следующей итерации цикла	continue
Тип чисел с <i>плавающей</i> точкой (вещественный)	double, float
Выполнение поразрядной (<i>побитовой</i>) операции исключающего ИЛИ	~
<i>Повторение</i> выполнения	do while
Получение адреса <i>подстроки</i>	strstr
Поиск <i>подстроки</i>	strstr
Поиск <i>подстроки</i> в строке	strstr
Получение <i>позиции</i> указателя в файле	fgetpos, ftell
Установка <i>позиции</i> указателя в файле	fsetpos, fseek
<i>Поиск</i> в большом массиве	bsearch
Выполнение двоичного <i>поиска</i>	bsearch

Выполняемое действие	Элемент C++
Поиск значения в массиве	bsearch
Объявление полиморфной функции	virtual
Получение адреса значения	&
Получение адреса подстроки	strstr
Получение времени выполнения программы	clock
Получение времени и даты	time
Получение данных по указателю	*
Получение даты	time
Получение длины строки	strlen
Получение дня недели	mktime, time
Получение индекса символа в блоке памяти	memchr
Получение информации о типе	dynamic_cast, typeid
Получение мантиссы и экспоненты	frexp
Получение номера текущей строки исходного файла	__LINE__
Получение ошибки файла	ferror
Получение параметров (командной строки)	main
Получение системного времени	time
Получение случайного числа	_rand
Получение позиции указателя в файле	fgetpos, ftell
Получение текущего исходного файла	__FILE__
Получение типа	typeid
Получение установленных параметров среды	getenv
Получение указателя на текущий объект	this
Поразрядное отрицание	~
Поразрядное (побитовое) И	&
Выполнение поразрядной (побитовой) операции исключающего ИЛИ	^
Выполнение поразрядных операций	AND(&), OR(), NOT(!)
Операторы для поразрядных операций (И, ИЛИ, НЕ, исключающее ИЛИ)	&, , ~, ^
Расширенные операторы работы с потоками	istream, ostream

Выполняемое действие	Элемент C++
Вывод в <i>поток</i>	cout
Вывод в <i>поток</i> ошибок	cerr
Ввод данных из <i>потока</i> ввода	cin
Ввод объекта из <i>потока</i>	istream
Ввод из файла как <i>потока</i>	ifstream
Возврат символа в <i>поток</i> ввода	ungetc
Объекты ввода/вывода для <i>потоков</i>	istream, ostream
Операторы работы с <i>потоками</i>	<<, >>
Объекты <i>потоков</i> ввода/вывода	fstream, ifstream, ofstream
Использование строки как объекта <i>потока</i> ввода	istrstream
Использование строки как объекта <i>потока</i> вывода	ostrstream
Использование <i>потока</i> файла ввода	ifstream
Установка буфера для операций над файлами (для <i>потока</i>)	setbuf, setvbuf
<i>Преобразование</i> времени по Гринвичу	gmtime
Сохранение типа (запрет <i>преобразования</i>)	explicit
<i>Преобразование</i> к символам верхнего регистра	toupper
<i>Преобразование</i> к символам нижнего регистра	tolower
<i>Преобразование</i> строки в число	atof, atoi, atol, strod, strtol, stroul
<i>Преобразование</i> строки в число с плавающей точкой	atof, strod
<i>Преобразование</i> типов данных	const_cast, dynamic_cast, reinterpret_cast, static_cast
<i>Преобразование</i> числа в строку цифр	sprintf
<i>Предотвращение</i> конфликта имен	namespace
Выполнение аварийного <i>прерывания</i> программы	abort
<i>Приведение</i> типа между полиморфными типами	dynamic_cast

Выполняемое действие	Элемент C++
<i>Приведение</i> типа между простейшими типами	static_cast
<i>Приведение</i> типа между связанными классами	static_cast
<i>Приведение</i> типа с отказом от модификатора const	const_cast
<i>Приведение</i> типа указателя к другому типу	reinterpret_cast
<i>Присваивание</i> с вычитанием	--
<i>Деление с присваиванием</i>	/=
<i>Присваивание</i> значения	=
<i>Сложение с присваиванием</i>	+=
<i>Умножение с присваиванием</i>	*=
<i>Компиляция при условии</i>	#if
<i>Проверка</i> выполнения равенства	==
<i>Проверка</i> массивов на равенство	memcmp
<i>Проверка</i> ложности значения	!
<i>Проверка</i> что больше	>
<i>Проверка</i> что меньше	<
<i>Проверка</i> на неравенство	!=
<i>Проверка</i> строк на равенство	strcmp
<i>Проверка</i> символа	is<cond>
<i>Продолжение</i> со следующей итерации цикла	continue
<i>Аварийный выход из программы</i>	abort
<i>Поддержка множественности версий программы</i>	#if
<i>Выполнение нормального выхода из программы</i>	exit
<i>Определение точки входа в программу</i>	main
<i>Старт программы</i>	main
<i>Регистрация (определение) функции завершения программы</i>	atexit
<i>Объявление пространства имен</i>	namespace
<i>Разрешение использования пространства имен</i>	using
<i>Использование символов из пространства имен</i>	using

продолжение ➤

Выполняемое действие	Элемент C++
Требование <i>прямого</i> доступа к памяти	volatile
Операторы <i>работы</i> с потоками	<<, >>
Расширенные операторы <i>работы</i> с потоками	istream, ostream
Проверка выполнения <i>равенства</i>	==
Проверка массивов на <i>равенство</i>	memcmp
Проверка строк на <i>равенство</i>	strcmp
Больше или <i>равно</i>	>=
Меньше или <i>равно</i>	<=
Объявление <i>разделяемых</i> переменных	extern
Получение <i>размера</i> массива	sizeof
Получение <i>размера</i> типа данных	sizeof
<i>Разрешение</i> изменений в константе	mutable
<i>Разрешение</i> доступа к закрытым членам	friend
<i>Разрешение</i> доступа к члену класса извне	public
<i>Разрешение</i> использования пространства имен	using
<i>Разрешение</i> использования идентификаторов из пространства имен	using
<i>Размещение</i> статических данных	static
Выделение памяти для <i>регистра</i>	register
Расположение переменной в <i>регистре</i> процессора	register
Преобразование к символам верхнего <i>регистра</i>	toupper
Преобразование к символам нижнего <i>регистра</i>	tolower
<i>Регистрация</i> (определение) функции завершения программы	atexit
Определение <i>регистровой</i> переменной	register
<i>Сброс</i> битов	bitwise AND (&)
<i>Сброс</i> буфера файла на диск	fflush
<i>Сброс</i> (очистка) флага ошибок	clearerr
<i>Сдвиг</i> битов вправо	>>
<i>Сдвиг</i> битов влево	<<

Выполняемое действие	Элемент C++
Сдвиг значений внутри блока памяти	memmove
Установка обработчика сигнала	signal
Создание сигнала	raise
Возврат символа в поток ввода	ungetc
Вывод символов в файл	fputc, putc
Вывод очередного символа в файл	fputc, putc
Получение индекса символа в блоке памяти	memchr
Запись (вывод) очередного символа в файл	fputc, putc
Чтение (ввод) очередного символа из файла	fgetc, getc
Ввод (чтение) очередного символа с консоли	getchar
Запись (вывод) очередного символа в файл	putchar
Чтение (ввод) очередного символа с клавиатуры	getchar
Проверка символа	is<cond>
Преобразование к символам верхнего регистра	toupper
Проверка символов на принадлежность к символам верхнего регистра	isupper
Преобразование к символам нижнего регистра	tolower
Проверка, что символ относится к символам нижнего регистра	islower
Тип символ	char
Удаление определения символа	#undef
Чтение символа с клавиатуры	getchar
Вычисление функции, обратной синусу	asin
Вычисление гиперболического синуса	sinh
Выполнение команды на уровне системы	system
Получение системного времени	time
Сложение	+
Инкремент (сложение значения с 1)	++

Выполняемое действие	Элемент C++
Сложение с присваиванием	+=
Выделение слов из строки	strtok
Генерация случайных чисел	rand
Начальная установка для случайных чисел	srand
Получение случайного числа	rand
Создание исключения	throw
Создание одного или более объектов	new
Создание сигнала	raise
Сортировка массива	qsort
Определение алгоритма быстрой сортировки	qsort
Сохранение адреса для longjmp	setjmp
Сохранение значения между вызовами функций	static
Сохранение значения переменной	const
Сохранение типа (запрет преобразования)	explicit
Определение списка констант	enum
Сравнение двух блоков памяти	memcmp
Сравнение массивов	memcmp
Сравнение строк	strcmp
Объявление переменной типа ссылка	&
Старт программы	main
Размещение статических данных	static
Возведение «e» в степень	exp
Возведение любого числа в степень	pow
Вычисление степени «e»	exp
Ввод значений из строки	sscanf
Вывод данных в строку с форматированием	sprintf
Ввод строки с клавиатуры	gets
Ввод данных из строки	sscanf
Использование строки как объекта потока ввода	istream
Использование строки как объекта потока вывода	ostream
Конкатенация строк	strcat

Выполняемое действие	Элемент C++
Копирование <i>строки</i>	strcpy
Выделение слов из <i>строки</i>	strtok
Выполнение операций над <i>строками</i>	str<op>
Комбинация <i>строк</i> (конкатенация)	strcat
Преобразование <i>строки</i> в целое	atoi, atol, strtol, stroul
Преобразование <i>строки</i> в число	atof, atoi, atol, strod, strtol, stroul
Проверка <i>строк</i> на равенство	strcmp
Поиск подстроки в <i>строке</i>	strstr
Получение длины <i>строки</i>	strlen
Получение номера текущей <i>строки</i>	__LINE__
Сравнение <i>строк</i>	strcmp
Чтение <i>строки</i> ввода	gets
Чтение данных из <i>строки</i>	sscanf
Чтение <i>строки</i> с клавиатуры	gets
Чтение <i>строки</i> с консоли	fgets
Объединение <i>строк</i>	strcat
Выполнение цикла по <i>счетчику</i>	for
Вычисление <i>тангенса</i>	tan
Вычисление гиперболического <i>тангенса</i>	tanh
Вычисление функции, обратной <i>тангенсу</i>	atan, atan2
Получение номера <i>текущей</i> строки исходного файла	__LINE__
Беззнаковые <i>типы</i> данных	union
Использование <i>типа</i> как параметра	template
Объявление нового имени <i>типа</i>	typedef
Объявление <i>типа</i> объединение	union
Объявление перечисляемого <i>типа</i>	enum
Приведение <i>типа</i> с отказом от модификатора const	const_cast
Булев <i>тип</i>	bool
<i>Тип</i> двухбайтовое (короткое) целое	short
<i>Тип</i> длинное целое	long
<i>Тип</i> короткое целое	short
<i>Тип</i> символ	char

Выполняемое действие	Элемент C++
<i>Тип</i> целое	int
<i>Тип</i> чисел с плавающей точкой (вещественный)	double, float
Объявление <i>типа</i> , вводимого пользователем	class, struct, union, typedef
Объявление переменной <i>типа</i> ссылки	&
Определение синонима для <i>типа</i>	typedef
Получение информации о <i>типе</i>	dynamic_cast, typeid
Получение размера <i>типа</i> данных	sizeof
Получение <i>типа</i>	typeid
Получение <i>типа</i> во время выполнения	dynamic_cast, typeid
Преобразование <i>типов</i> данных	const_cast, dynamic_cast, reinterpret_cast, static_cast
Приведение <i>типа</i> между простейшими типами	static_cast
Приведение <i>типа</i> между полиморфными типами	dynamic_cast
Приведение <i>типа</i> между связанными классами	static_cast
Приведение <i>типа</i> указателя к другому типу	reinterpret_cast
Установка <i>точки</i> возврата для longjmp	setjmp
Определение <i>точки</i> входа в программу	main
Увеличение размера выделенного блока памяти	realloc
Удаление дробной части из числа	modf
Удаление объекта	delete
Удаление определения символа	#undef
Удаления файла с диска	remove
Доступ к членам объекта через <i>указатель</i>	->
Изменение типа <i>указателя</i>	reinterpret_cast
Объявление <i>указателя</i>	*
Операторы <i>указателя</i> на член	* ->*
Получение данных по <i>указателю</i>	*
Получение позиции <i>указателя</i> в файле	fgetpos, ftell

Выполняемое действие	Элемент C++
Получение <i>указателя</i> на текущий объект	this
Установка позиции <i>указателя</i> файла	fseek, fsetpos
Установка позиции <i>указателя</i> в файле	fsetpos, fseek
Умножение	*
Умножение с присваиванием	*=
Если не выполняется <i>условие</i>	else
Условная компиляция	#if
Определение <i>условия</i> конца файла	feof
Установка адреса для длинного перехода	setjmp
Установка битов	bitwise OR ()
Установка буфера для файловых операций	setbuf, setvbuf
Установка возвращаемого значения	return
Установка значений в блоке памяти	memset
Установка начала генерации случайных чисел	srand
Установка обработчика сигнала	signal
Установка позиции указателя файла	fseek, fsetpos
Установка точки возврата для longjmp	setjmp
Установка указателя файла	fseek, tsetpos
Ввод из <i>файла</i> как потока	ifstream
Ввод строки из <i>файла</i>	fgets
Включение <i>файла</i> заголовков	#include
Вывод двоичных данных в <i>файл</i>	fwrite
Вывод строки в <i>файл</i>	fputs
Вывод символов в <i>файл</i>	fputc, putc
Вывод очередного символа в <i>файл</i>	putchar
Выполнение очистки буфера <i>файла</i>	fflush
Выполнение форматированного ввода из <i>файла</i>	fscanf
Закрытие <i>файла</i>	fclose
Запись двоичных данных в <i>файл</i>	fwrite
Запись очередного символа в <i>файл</i>	fputc, putc
Закрытие и открытие <i>файла</i>	freopen
Вывод времени в <i>формате</i>	strftime

Выполняемое действие	Элемент C++
Изменение имени <i>файла</i>	rename
Использование потока <i>файла</i> ввода	ifstream
Определение условия конца <i>файла</i>	feof
Отказ от буферизованной работы с <i>файлами</i>	setbuf, setvbuf
Открытие <i>файла</i>	fopen
Открытие <i>файла</i> заново	freopen
Очистка буфера <i>файла</i>	fflush
Переименование <i>файла</i>	rename
Получение временного <i>файла</i>	tmpfile
Получение ошибки <i>файла</i>	ferror
Получение текущего исходного <i>файла</i>	__FILE__
Получение позиции указателя в <i>файле</i>	fgetpos, ftell
Удаление <i>файла</i> с диска	remove
Установка буфера для <i>файловых</i> операций	setbuf, setvbuf
Установка позиции указателя <i>файла</i>	fseek, fsetpos
Чтение двоичных данных из <i>файла</i>	fread
Чтение двоичных данных из <i>файла</i>	fread
Чтение очередного символа из <i>файла</i>	fgetc, getc
Чтение строки из <i>файла</i>	fgets
Чтение <i>файла</i> заголовков	#include
Форматированный вывод в <i>файл</i>	fprint
Форматированный ввод в <i>файл</i>	fscanf
Форматированный вывод в <i>файл</i>	fprint
Установка позиции указателя в <i>файле</i>	fsetpos, fseek
Форматированный вывод в <i>файл</i>	fprint
Вывод данных в строку с форматированием	sprintf
Форматированный вывод на стандартное устройство вывода	printf
Форматированный ввод в <i>файл</i>	fscanf
Форматированный вывод в <i>файл</i>	fprint
Формирование возвращаемого значения	return
Возврат в вызвавшую <i>функцию</i>	return
Вставка текста <i>функции</i> в программу	inline
Выполнение выхода из <i>функции</i>	return

Выполняемое действие	Элемент C++
Вычисление <i>функции</i> , обратной косинусу	acos
Вычисление <i>функции</i> , обратной синусу	asin
Вычисление <i>функции</i> , обратной тангенсу	atan, atan2
Выход из <i>функции</i>	return
Компиляция <i>функции</i> как встроенной	inline
Объявление виртуальной <i>функции</i> (функции позднего связывания)	virtual
Объявление полиморфной <i>функции</i>	virtual
Объявление шаблона класса или <i>функции</i>	template
Определение и объявление <i>функции</i> оператора	operator
Определение макро <i>функции</i> (макроса)	#define
Определение шаблона класса или <i>функции</i>	template
Переход между <i>функциями</i>	longjmp
Регистрация (определение) <i>функции</i> завершения программы	atexit
Выполнение <i>цикла</i> определенное количество раз	for
Выполнение <i>цикла</i> по счетчику	for
Вычисление <i>целой</i> части числа	modf
Переход к выполнению следующей итерации <i>цикла</i>	continue
Продолжение со следующей итерации <i>цикла</i>	continue
Тип <i>целое</i>	int
Тип короткое <i>целое</i>	short
Преобразование <i>числа</i> в строку цифр	sprintf
Преобразование строки в <i>число</i>	atof, atoi, atol, strtod, strtol, strtoul
Доступ к <i>членам</i> объекта	.
Доступ к <i>члену</i> через указатель	->
Ограничение доступа к <i>членам</i>	private, protected
Операторы указателя на <i>член</i>	* ->*

продолжение ↗

Выполняемое действие	Элемент C++
Разрешение доступа к внутренним членам	Friend
Чтение данных с клавиатуры	Scanf
Чтение данных с консоли	Scanf
Чтение данных из строки	Sscanf
Чтение двоичных данных из файла	Fread
Чтение очередного символа с консоли	Getchar
Чтение очередного символа из файла	fgetc, getc
Чтение символа с клавиатуры	getchar
Чтение строки ввода	gets
Чтение строки с клавиатуры	gets
Чтение строки с консоли	gets
Чтение строки из файла	fgets
Чтение файла заголовков	#include
Объявление шаблона класса или функции	template
Получение ширины типа данных	sizeof
Получение мантиссы и экспоненты	frexp
Вывод данных на экран	putchar, puts, printf
Обращение к элементу массива	[]

Алфавитный указатель

Символы

--, 108
!=, 114
%=, 108
%ld, 326
%s, 323
&, 359
&&, 111
&=, 108
*/ , 300
*=, 108
++, 109
+=, 108
--, 109
/*, 300
//, 300
/=, 108
<<, 108
<<=, 108
<=, 114
==, 114
>=, 114
>>, 108
>>=, 108
?, 110
^=, 108
|=, 108
~, 108

C

callback-функция, 80
char, 320

D

derived, 389

E

Esc-последовательность, 293
external, 304

H

header files, 296

I

if, 304

O

overloading, 356, 366

S

static, 303
stream, 329

V

virtual, 407

W

while, 304

A

абсолютное значение, 210, 221
абстрактный класс, 28, 420
абстрактный тип данных, 28
абстракция данных, 29
аварийное завершение
 программы, 210
агрегат, 29
анонимные объединения, 30

аргумент, 31
арифметические операции, 298
арифметический тип
 данных, 84
арккосинус, 211
арксинус, 211
арктангенс, 212, 213
ассемблер, 128
ассоциативность, 31
атрибут доступа, 342

Б

базовый класс, 32, 390, 405
библиотека
 ввода / вывода, 178
 математических
 функций, 180
 строковая, 179
 типов символов, 179
 функций даты и
 времени, 180
 функций памяти, 180
 языка C, 178
бинарные операции, 299, 308
бинарный оператор, 382
бинарный режим, 279
битовое поле, 33
битовые операторы, 53
булев тип данных, 88
булево выражение, 33

В

ввод
 данных, 242, 252
 из строки, 252
 признак окончания, 298
 с клавиатуры, 242, 297
 строки, 333
«венгерская» схема, 42
версии программы, 167
виртуальная функция, 34,
 155, 409

виртуальный файл
 заголовков, 79
вложенная инструкция, 35
вложенность, 35
внешняя переменная, 135
время
 компиляции, 176
 формат, 253
встроенная функция
 создание, 35

Вывод

 в строку, 251
 в файл, 236
 данных, 233, 235
 на экран, 186
 с форматированием, 233
 символа, 236, 237
 строки, 186, 237
 ширина поля, 235
вызов конструктора, 364
выражение, 36
 простое, 299
 составное, 299

Г

генератор случайных
 чисел, 240
гиперболический косинус, 219
гиперболический синус, 250
гиперболический тангенс, 258
главная функция, 36
глобальная переменная, 303

Д

дамп, 208
данные
 абстракция, 29
 ввод из строки, 252
 ввод с клавиатуры, 242
 доступ по указателю, 52
 форматированный
 вывод, 233

данные (*продолжение*)

члены, 37, 342

чтение и запись

на консоли, 184

чтение из файла, 203

дата

компиляции, 175

формат, 253

двойной слэш, 300

декорирование, 37

декремент, 109, 307

деление

длинных целых, 228

по модулю, 111

деструктор, 37, 351

дизъюнкция, 110

динамический объект

выделение памяти, 104

удаление, 105

директива, 38

#define, 38

#elif, 164

#else, 164

#error, 165

#if, 166

#ifdef, 169

#ifndef, 169

#include, 296

#line, 171

#pragma, 172

#undef, 173

директивы

определение, 296

чтение файла

заголовков, 296

дочерний класс, 69, 391

3

заголовок функции, 302

запись

в определенном

формате, 262

в файл, 283

строки в память, 288

запятая, 108

знакорасширение, 40

значение по умолчанию, 41

И

идентификатор, 41, 161

иерархия наследования, 395

именующее выражение, 43

имя

класса, 344

массива, 54

текущего файла, 175

инверсия, 108

индекс массива, 54

инициализация

массива, 30

сложных типов данных, 29

инкапсуляция, 43, 65, 348

инкремент, 109, 307

инструкции

безусловного перехода, 137

вложенные, 35

выхода из цикла, 128

исполняемые, 46

метки, 54

обработки

исключений, 129, 150, 151

объявления

переменных, 294

прерывания итераций

цикла, 132

присваивания, 44

пустые, 70

составные, 74

условного перехода, 147

условные, 78, 137

цикла с постусловием, 132

цикла с предусловием, 157

инструкция, 44

break, 128

continue, 132

do, 132

else, 133

инструкция (*продолжение*)

for, 135
 goto, 137
 if, 137
 return, 143, 301, 312
 switch, 147
 throw, 150
 try, 151
 while, 157

интерфейс класса, 348

исключение, 45, 435

исполняемая инструкция, 46

К

квадратный корень, 251

клавиатура

чтение символов, 185, 227
 чтение строки, 227
 чтение чисел, 186

класс, 46, 341

fstream, 274
 ifstream, 275
 ios, 268
 istream, 266, 279
 istrstream, 283
 logic_error, 435
 ofstream, 283
 ostream, 286
 ostrstream, 288
 runtime_error, 435
 strstream, 288

классы

абстрактный, 28, 420
 атрибуты доступа, 342
 базовый, 32, 390
 ввода / вывода, 270
 инициализация
 объектов, 355
 исключений, 435
 исполнительная часть, 348
 методы создания, 399
 новый тип, 270
 область видимости, 342

классы (*продолжение*)

обобщение, 377
 объявление
 производного, 391
 памяти, 47
 предок, 32
 производный, 68
 производный от
 производного, 395
 родительский, 32
 создание
 без наследования, 397
 создание в виде
 оболочки, 397

ключевое слово, 47, 128

asm, 128
 auto, 128
 break, 128
 case, 128
 catch, 129
 class, 341
 const, 361
 continue, 132
 default, 132
 do, 132
 else, 133
 explicit, 134
 extern, 135, 304
 for, 135
 friend, 369
 goto, 137
 if, 137
 inline, 138, 354
 main, 139
 mutable, 139
 namespace, 140
 operator, 141
 private, 142
 protected, 142
 public, 142, 391
 register, 142
 return, 143
 static, 143, 303
 struct, 146, 354

ключевое слово (*продолжение*)

switch, 147
template, 147
this, 150, 371, 374
throw, 150
try, 151
typedef, 153
union, 154, 354
using, 155
virtual, 155, 409
void, 156
volatile, 156
while, 157

команда

на языке ассемблера, 128
операционной системы, 257

командная строка, 168

комментарий, 47, 300

компиляция

на альтернативных
условиях, 164
остановка, 165
препроцессорная фаза, 159
при выполнении
условия, 166
при невыполнении
условия, 164
текущая дата, 175
текущее время, 176
условная, 79

конец

строки, 73
условной компиляции, 165

консоль, 184

консоль ошибок, 271

константа, 48

беззнаковая, 87
в виде беззнакового
целого, 93
в восьмеричной системе
счисления, 86
в шестнадцатеричной
системе счисления, 86

константа (*продолжение*)

литерал, 52
с плавающей точкой, 87
символическая, 48, 160
символьная, 321
тип по умолчанию, 97
числовая, 48

константная переменная, 131

константный указатель, 131

конструктор, 49, 351, 355

fstream, 274
ifstream, 275
istream, 283
ofstream, 283
ostream, 288
stringstream, 289

конструкторы

базового класса, 50,
404, 405
вызов, 364
копирования, 51, 355, 357,
358, 381
неявный, 51
перегрузка, 356
по умолчанию, 51, 357
скрытый, 51, 357

конъюнкция, 110

корень квадратный, 251

«короткая схема» логики, 111

косвенная адресация, 312

косвенный вызов функции, 409

косвенный доступ, 52

косинус, 219

Л

лексема, 200

литерал, 52

логарифм

десятичный, 230
натуральный, 230

логические

операторы, 52, 110
операции, 308

логическое

И, 110

ИЛИ, 110

отрицание, 110

локальная переменная, 303

М

макрос, 53, 173

__cplusplus, 174

__DATE__, 174

__FILE__, 174, 175

__LINE__, 174, 175

__TIME__, 174, 176

va_arg, 182

va_end, 182

va_start, 182

обработки списков

аргументов, 182

макрофункция, 160

манипулятор

endl, 262

hex, 262

oct, 262

манипуляторы

ввода/вывода, 266

мантисса, 226

массив, 54, 315

двухмерный, 315

имя, 54

инициализация

элементов, 319

нумерация элементов, 315

размещение в памяти, 317

символов, 76

сортировка, 237

сравнение элементов, 216

трехмерный, 316

меню приложения, 413

мета-данные, 107

метка, 54

метод, 81

многострочный

комментарий, 48

множественное

наследование, 55

модификатор

*, 244

const, 116, 131

explicit, 134

extern, 135

h, 244

inline, 138

l, 244

mutable, 139

private, 142

protected, 142

public, 142

register, 142

static, 143

volatile, 156

модификаторы

доступа, 142

типа, 143

формата, 235, 244

модуль, 303

Н

наследование, 56, 389

множественное, 55

оптимизация, 396

необрабатываемое

исключение, 45

неравенства, 114

неявный конструктор

копирования, 51

номер текущей строки, 175

О

область

видимости переменной, 302

действия переменной, 56

изменения чисел, 86

определения в производном

классе, 394

памяти, 312

- обратный слэш, 293
- объединение, 354
- объект, 57
 - cerr, 271
 - cin, 272
 - clog, 273
 - cout, 273
 - ifstream, 277
- объектная ориентация, 57
- объектно-ориентированное программирование, 58
- объекты
 - динамическое создание, 347
 - класса, 83, 345
 - копирование, 51
 - нулевой длины, 277
 - область определения, 57
 - создание из класса, 134
 - удаление, 359
- объявление, 58
 - виртуальной
 - функции, 155, 409
 - дружественных функций
 - или классов, 136
 - закрытых членов
 - класса, 142
 - защищенных членов
 - класса, 142
 - локальной переменной, 128
 - открытых членов
 - класса, 142
 - переменной, 294
 - переменной, изменяемой извне, 156
 - переменных в регистровой памяти, 142
 - родового типа, 82
 - статической функции, 145
 - функции, 296
- одновременный ввод и вывод, 274
- операнд, 60, 98
 - определение типа, 105
- операнд (*продолжение*)
 - размер в байтах, 103
 - сравнение, 113
 - увеличение значения, 109
 - уменьшение значения, 109
- оператор, 60, 98
 - ., 108
 - ==, 107
 - >, 347
 - #, 176
 - ##, 176
 - %, 111
 - %=, 107
 - &, 53, 359
 - &&, 111
 - &=, 107
 - (type), 117
 - *=, 107
 - ++, 109
 - +=, 107
 - , 109
 - /=, 107
 - <<=, 107
 - =, 107
 - >>=, 107
 - ?, 98, 110
 - ^=, 107
 - |, 53
 - |=, 107
 - ~, 53
 - const_cast, 118
 - defined, 176
 - delete, 105, 326
 - dynamic_cast, 120
 - new, 104, 326
 - reinterpret_cast, 116, 122
 - sizeof, 103
 - static_cast, 116, 124
 - typedef, 153
 - typeid, 105
- операторы
 - ассоциативность, 31
 - бинарные, 98, 382
 - ввода, 330

операторы (*продолжение*)

- ввода / вывода, 263
- вывода, 263, 330
- выделения элемента
 - массива, 386
- вызова функции, 387
- декремента, 109
- деления по модулю, 111
- имя функции, 382
- инкремента, 109
- области определения, 394
- ограничения при
 - перегрузке, 388
- отношения, 113
- перегрузка, 141, 263, 265, 366, 382
- переопределение, 141, 263, 265
- поразрядные, 53, 108
- порядок
 - выполнения, 31, 267
- постфиксные, 386
- преобразования
 - объектов, 116
- преобразования типов, 116
- преобразования
 - указателей, 116
- препроцессора, 176
- префиксные, 386
- приведения типов, 116, 117
- приоритет, 31, 99
- присваивания, 61, 107, 372, 381, 384
 - с вычитанием, 385
 - со сложением, 385
- простые, 108
- сдвига
 - влево, 267
 - вправо, 265
- сложные, 107
- ссылки, 359
- тринарные, 98
- указатели, 111
- унарные, 98, 383

операторы (*продолжение*)

- условные, 98, 110
- функции, 366
- операции
 - арифметические, 298
 - бинарные, 299, 308
 - логические, 308
 - над битами, 308
 - присваивания, 308
 - расширенные, 308
 - сравнения, 298
 - унарные, 299, 308
- определение, 62
 - видимости, 114
 - встраиваемой функции, 138
 - идентификатора, 161, 164
 - оператора инкремента, 386
 - оператора присваивания со сложением, 385
 - переменной, 62
 - функции, 62, 72, 301
 - функции в классе, 81
- отличия C и C++, 426
- очистка буфера, 225
- ошибка
 - буферизованный вывод, 273
 - выполнения
 - программы, 435
 - обработчик сигналов, 249
 - регистрация обработчика сигналов, 248
 - сигнал возникновения, 239
 - сообщение, 232
 - устройство вывода, 271

П

память

- выделение, 218, 231
- динамическое
 - распределение, 324
- изменение выделяемого размера, 240

- память (*продолжение*)
 - освобождение, 225
 - статическое
 - выделение, 143
 - распределение, 323
- параметр
 - защита от изменений, 361
 - по значению, 313
 - по ссылке, 313
 - способы передачи, 360
 - среды, 229
 - текущей локализации, 246
 - фактический, 31
 - формальный, 31
- перегрузка, 63
 - конструктора, 356
 - операторов, 141, 366, 382
 - функций, 63, 356
- переименование папки, 241
- переменная, 64
 - в определении класса, 37
 - внешняя, 304
 - выделение памяти, 294
 - глобальная, 303
 - локальная, 303
 - область видимости, 302
 - объявление, 294
 - объявленная как `const`, 49
 - присваивание значения, 294
 - размещение в памяти, 47
 - с не изменяющимся значением, 131
 - сохранение имени и типа, 37
 - ссылки, 74
 - статическая, 303
 - указатель
 - на описание, 227
- переопределение встроенного оператора, 141
- переход на следующую строку, 262
- переходы между функциями, 230, 245
- перечисление, 64
- подкласс, 69
- позднее связывание, 408
- поиск в массиве, 216
- показатель степени, 226
- поле бит, 33, 309
- полиморфизм, 65
- полиморфное приведение типа, 120
- поразрядный оператор, 108
- постфикс, 307
- поток
 - ввода, 329
 - вводимого файла, 337
 - ввода / вывода, 188
 - вывода, 329
- потомок, 391
- прагма, 66
- предок, 391
- представление чисел
 - с плавающей точкой, 95
- преобразование типов, 81, 125
- препроцессорная обработка, 38
- префикс, 307
- приведение типа, 66
 - полиморфное, 120
- приложение, 66
- примитивный тип, 69
- принцип позиционной независимости, 293
- приоритет, 67, 99
- присваивание, 107
- программа
 - аварийное завершение, 210
 - аккуратное завершение, 220
 - ввода строки, 334
 - вывода каталога, 257
 - вывода содержимого файла, 278
 - вывода строки, 292
 - вычисления
 - факториала, 143
 - завершение
 - при условии, 212

программа (*продолжение*)
 копирования текстового
 файла, 207
 начало выполнения, 139
 открытия файла, 224
 перегрузки операторов
 сдвига, 265
 переименования файла, 242
 перемещения файла, 242
 получения дня недели, 196
 представления в различных
 форматах, 267
 сборка с учетом типов, 32
 создание, 294
 создания меню
 приложения, 413
 тело, 98
 функция завершения, 213
 продвижение, 68
 производный класс, 68
 простейший тип данных, 69
 простое выражение, 299
 простой
 оператор, 108
 тип данных, 84
 пространство имен, 69
 прототип, 70
 функции, 301
 пустая инструкция, 70

Р

разыменованье указателя, 71
 расширенная операция
 присваивания, 308
 реализация, 72
 рекурсия, 72
 родительский класс, 32, 391

С

сдвиг влево, 108
 сдвиг вправо, 108
 сигнатура, 50, 73

СИМВОЛ

&, 297, 359
 ;, 293
 <<, 331
 >>, 331
 \, 293
 0, 321
 вывод в поток, 237
 вывод в файл, 236
 новой строки, 73
 нуль-терминатор, 321
 перевод
 в верхний регистр, 259
 в нижний регистр, 259
 проверка значения, 189
 управляющий, 321
 чтение
 из файла, 226
 с клавиатуры, 185
 с консоли, 227
 символическая константа, 48,
 131, 160
 создание, 160
 символическое имя, 173
 синтаксис препроцессора, 159
 синус, 250
 системная команда, 257
 скрытый конструктор, 357
 сложный оператор
 присваивания, 107
 сложный тип данных, 29, 73
 создание
 меню приложения, 413
 переменной, 62
 сообщение об ошибке, 165, 232
 сортировка массива, 237
 составная инструкция, 44, 74
 составное выражение, 299
 сохранение констант, 96
 спецификатор
 %d, 295
 %ld, 326
 %s, 323
 auto, 128

спецификатор (*продолжение*)

- class, 130
- enum, 133
- friend, 136
- namespace, 140
- operator, 141
- private, 400
- protected, 400
- public, 401
- virtual, 155
- void, 156

спецификаторы

- базовых типов, 156
- доступа, 136, 400
- перечисляемых типов, 133
- типов, 130
- формата, 234, 242, 253

список аргументов переменной
длины, 182

ссылка

- вперед, 75
- на класс
 - ввода/вывода, 271
- на левый операнд, 375
- на объект, 51, 374
- на потоковый объект, 271
- на функцию, 408
- переменная, 74

статическая переменная, 303

статическая функция, 304

стек, 312

строка, 76

- вывод, 186, 237, 251
- поиск, 200
- преобразование
 - в вещественное, 214
 - в длинное
 - целое, 216, 256
 - в целое, 215
 - в число, 255, 257
- принудительный
 - вывод, 187
- системная команда, 257
- создание указателя, 200

строка (*продолжение*)

- текущий номер, 175
- чтение из текстового
 - файла, 207
- чтение с консоли, 227

строковый комментарий, 47

строковый литерал, 49

структура, 354

- управляющая, 35

суффикс

- F, 96
- L, 94
- U, 93, 94

Т

таблица указателей

- виртуальных функций, 422

тангенс, 258

текстовый режим, 279

тело

- программы, 98
- функции, 301

тип

- char, 320
- операнда, 105
- оператора, 98

тип данных, 84

- bool, 88
- char, 89
- double, 96
- float, 95
- int, 92
- long, 94
- long double, 97
- short, 94
- signed char, 91
- unsigned char, 91
- unsigned int, 93
- unsigned long, 95
- unsigned short, 94
- wchar_t, 90, 97

типы данных

- абстрактный, 28
- беззнаковый, 32

типы данных (*продолжение*)

- битовое поле, 33
- булев, 34
- инициализация, 29
- определение нового, 153
- по умолчанию, 85
- сложный, 29, 73

тождественное равенство, 114

тринарный оператор, 98, 110

у

удаление

- модификатора `const`, 116
- объекта из памяти, 38, 351
- программного кода, 48

указатель, 77, 311

- базового типа, 407

- в файле, 204

- константный, 131

- на класс, 407

- на массив, 319

- на объект, 150, 347, 374

- на строку, 200

- на функцию, 77

- на член объекта, 113

- разыменование, 71

унарные операции, 299, 308

унарный оператор, 383

управляющая

- последовательность, 78,

- 90, 293

управляющая структура, 35, 78

управляющий символ, 90, 293

условная компиляция, 79, 167

- конец, 165

условный оператор, 98, 110

Ф

файл

- бинарный, 206, 207

- виртуальный, 79

- заголовок, 430

файл (*продолжение*)

- временный, 258

- глубина вложенности, 170

- двоичный, 223

- заголовков, 79, 296

- запись, 283

- обработка ошибок, 205

- открытие, 205, 222, 275, 284

- открытие для чтения и записи, 337

- переименование, 241

- содержание, 79

- текстовый, 223

- чтение, 203, 226, 278

фактический параметр, 31

флаги

- ввода / вывода, 268

- снятие, 270

- текущая установка, 270

- установка, 269

- установка нескольких, 270

- форматирования, 262, 268

формальный параметр, 31

формат даты и времени, 253

функции

- аргумент по умолчанию, 41

- базового класса, 391

- без аргументов, 51

- бинарного оператора, 382

- ввода, 281

- ввода строки, 333

- ввода / вывода, 279

- виртуальные, 155, 409

- включение в проект, 297

- встраиваемые, 35, 138, 353

- входа в программу, 302

- вывода строки, 333

- вызов, 301

- вызывающие саму себя, 72

- выход, 301

- главная, 36

- даты и времени, 193

- деления целого

- на целое, 220

функции (*продолжение*)

- динамического выделения памяти, 324
- друзья, 369
- завершения программы, 213
- инициализации класса, 49
- инициализации объекта, 51
- инструкция возврата, 143
- класса, 344
- класса ifstream, 277
- консольного
 - ввода/вывода, 183
- не возвращающие значения, 80
- не содержащие кода, 82
- обработки списков аргументов, 182
- обратного вызова, 80
- оператора
 - delete, 387
 - new, 387
 - выделения элемента массива, 387
 - присваивания, 384
- определение в классе, 81
- определения количества введенных символов, 279
- определения состояния ввода/вывода, 280
- перегрузка, 63, 265
- переопределения встроенного оператора, 141
- переходы, 230, 245
- поиска в массиве, 216
- преобразования типа, 81
- присваивания, 384
- проверки категории символов, 189
- проверки корректности выделения памяти, 325
- работы с блоками памяти, 191

функции (*продолжение*)

- рекурсивные, 72
- строковые, 197, 323
- типа void, 300
- указателя в файле, 204
- унарного оператора, 383
- файлового
 - ввода/вывода, 201
- формата вывода, 286
- чистые
 - виртуальные, 82, 414
- члены, 81, 342

функция, 80, 300

- abort, 210
- abs, 210
- acos, 211
- asin, 211
- assert, 212
- atan, 212
- atan2, 213
- atexit, 213
- atof, 214
- atoi, 215
- atol, 216
- bsearch, 216
- calloc, 218
- ceil, 218
- clearerr, 205
- clock, 193
- cos, 219
- cosh, 219
- ctime, 194
- difftime, 194
- div, 220
- exit, 220
- exp, 221
- fabs, 221
- fclose, 202
- feof, 204
- ferror, 205
- fflush, 187, 203
- fgetc, 203
- fgetpos, 204

функция (продолжение)

fgets, 185, 203, 207
flags, 270
floor, 221
fmod, 222
fopen, 202, 222, 335
fprintf, 203
fputc, 203
fputs, 188, 203, 207
fread, 204
free, 225, 325
freopen, 202
frexp, 226
fscanf, 204
fseek, 204
fsetpos, 204
ftell, 204
fwrite, 203, 209
gcount, 279
getc, 204, 226
getchar, 184, 185, 227
getenv, 227
getline, 272, 339
gets, 184, 227
gmtime, 194
isalnum, 189
isalpha, 189
iscntrl, 189
isdigit, 189
isgraph, 190
islower, 190
isprint, 190
ispunct, 190
isspace, 190
isupper, 190
isxdigit, 190
labs, 228
ldexp, 228
ldiv, 228
localeconv, 229
localtime, 194
log, 230
log10, 230
long unsigned coreleft, 325

функция (продолжение)

longjmp, 230
main, 139, 302
malloc, 324
mktime, 194
modf, 232
perror, 184, 232
pow, 233
printf, 184, 233, 234, 235
putc, 203, 236
putchar, 184, 237
puts, 185, 186, 237, 333
qsort, 237
raise, 239
rand, 240
rdstate, 280
read, 279
realloc, 240
rename, 241
rewind, 205
scanf, 185, 186, 242, 244, 297
setbuf, 205
setf, 269
setjmp, 245
setlocale, 246
setvbuf, 205
signal, 248
sin, 250
sinh, 250
sprintf, 251
sqrt, 251
srand, 252
sscanf, 252
strcat, 198
strchr, 198
strcmp, 198
strcoll, 198
strcpy, 198
strcspn, 198
strerror, 198
strftime, 194, 253
strlen, 198
strncat, 198

функция (продолжение)

strncmp, 198
strncpy, 198
strpbrk, 199
strrchr, 199
strspn, 199
strstr, 198
strtod, 255
strtok, 199
strtol, 256
strtoul, 257
strxfrm, 199
system, 257
tan, 258
tanh, 258
time, 194
tmpfile, 202, 258
tolower, 259
toupper, 259
ungetc, 204
unsetf, 270

Ц

цепочка преобразования
типов, 126

цикл

с постусловием, 132
с предусловием, 157

Ч

число

абсолютное значение
длинного целого, 228
возведение в степень, 233
деление длинного
целого, 228
округление до большего
целого, 218
округление до меньшего
целого, 221
остаток от деления, 222

число (продолжение)

разбиение на целую
и дробную часть, 232
с плавающей точкой
двойной точности, 96
с плавающей точкой
экстра-точности, 97
случайное, 240, 252
создание по мантиссе
и показателю
степени, 228
формат по умолчанию, 332
чтение с клавиатуры, 186
числовая константа, 48
чистая виртуальная
функция, 82, 414
член, 82
чтение
в определенном
формате, 262
из файла, 278
символа
из файла, 226
с клавиатуры, 185
с консоли, 227
строки
в памяти, 288
с консоли, 227
числа с клавиатуры, 186

Ш

шаблон, 82
описание, 147
ширина поля, 235

Э

экземпляр, 83
класса, 83
экспонента, 221
экспоненциальное
представление, 95

УВАЖАЕМЫЕ ГОСПОДА! ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» ПРИГЛАШАЕТ ВАС К ВЗАИМОВЫГОДНОМУ СОТРУДНИЧЕСТВУ. ЗА ДОПОЛНИТЕЛЬНОЙ ИНФОРМАЦИЕЙ ОБРАЩАЙТЕСЬ В НАШИ ПРЕДСТАВИТЕЛЬСТВА, КОТОРЫЕ НАХОДЯТСЯ ПО СЛЕДУЮЩИМ АДРЕСАМ:

Россия, г. Москва

Представительство издательства «Питер», м. «Калужская»,
ул. Бутлерова, д. 176, оф. 207 и 240,
тел./факс (095) 234-38-15, 333-15-73.
E-mail: sales@piter.msk.ru

Россия, г. Санкт-Петербург

Представительство издательства «Питер»,
м. «Электросила», ул. Благодатная, д. 67,
тел. (812) 327-93-37, 294-54-65.
E-mail: sales@piter.com

Украина, г. Харьков

Представительство издательства «Питер»,
тел. (0572) 23-75-63, факс (0572) 14-96-09.
Почтовый адрес: 310093, г. Харьков, а/я 9130.
E-mail: piter@tender.kharkov.ua

Украина, г. Киев

Филиал Харьковского представительства
издательства «Питер», тел./факс (044) 490-35-68,
тел. 490-35-69. Почтовый адрес: 04073, г. Киев,
пр. Красных Казаков, д. 6, корп. 1.
E-mail: office@piter-press.kiev.ua

Беларусь, г. Минск

Представительство издательства «Питер»,
тел./факс (0172) 16-00-06. Почтовый адрес:
220012, г. Минск, ул. Кедышко, д. 19, а/я 104.
E-mail: piterbel@carry.nsby.by

**КАЖДОЕ ИЗ ЭТИХ ПРЕДСТАВИТЕЛЬСТВ РАБОТАЕТ
ПО ЕДИНОМУ СТАНДАРТУ ИЗДАТЕЛЬСКОГО ДОМА
«ПИТЕР».**

КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» ВЫ МОЖЕТЕ ПРИОБРЕСТИ ОПТОМ И В РОЗНИЦУ У НАШИХ РЕГИОНАЛЬНЫХ ПАРТНЕРОВ.

Башкортостан

Уфа, «Азия», ул. Зенцова, д. 70 (оптовая продажа),
маг. «Оазис», ул. Чернышевского, д. 88,
тел./факс (3472) 50-39-00.
E-mail: asiaufa@ufanet.ru

Дальний Восток

Хабаровск, «Мирс», тел. (4212) 22-74-58, факс 22-73-30.
Почтовый адрес: 680000, г. Хабаровск,
ул. Ким-Ю-Чена, д. 21.
E-mail: postmaster@bookmirs.khv.ru

Владивосток, «Приморский Торговый Дом Книги»,
тел./факс (4232) 23-82-12. Почтовый адрес: 690091,
ул. Светланская, д. 43.
E-mail: bookbase@mail.primorye.ru

Хабаровск, «Книжный мир», тел. (4212) 32-85-51,
факс 32-82-50. Почтовый адрес: 680000, г. Хабаровск,
ул. Карла Маркса, д. 37.

Европейские регионы России

Архангельск, «Дом Книги», тел. (8182) 64-00-56,
факс 65-41-34. Почтовый адрес: 163061, пл. Ленина, д. 3.
E-mail: book@atnet.ru

Калининград, «Пресса», тел. (0112) 53-66-97, факс
53-63-87. Почтовый адрес: 236040, ул. Подполковника
Иванникова, д. 3а.
E-mail: book@pressa.gazinter.net

Краснодар, «Когорта», тел./факс (8612) 62-54-97.
Почтовый адрес: 350033, г. Краснодар, ул. Ленина, д. 101.

Северный Кавказ

Ессентуки, «Россы», ул. Октябрьская, 424,
тел./факс (86534) 6-93-09.
Почтовый адрес: 357600, г. Ессентуки, а/я 80.
E-mail: rossy@kmv.ru

Сибирь

Братск, «Прометей», тел./факс (3953) 43-18-76.
Почтовый адрес: 665708, г. Братск, ул. Кирова, д. 35.

Иркутск, «Продалить», тел. (3952) 59-13-70, факс 51-23-31.
Почтовый адрес: 664031, г. Иркутск,
ул. Байкальская, д. 172, а/я 1397.
E-mail: prodalit@irk.ru; http://www.prodalit.irk.ru

Иркутск, «Антей-книга», тел./факс (3952) 34-69-38.
Почтовый адрес: 664003, ул. Карла Маркса, д. 20.
E-mail: antey@irk.ru

Красноярск, «Книжный Мир», тел./факс (3912) 27-39-71.
Почтовый адрес: 660049, г. Красноярск, пр. Мира, д. 86.
E-mail: book-world@public.krasnet.ru

Нижневартовск, «Дом книги», тел. (3466) 23-59-50, факс
23-27-14. Почтовый адрес: 626440, пр. Победы, д. 12.

Новосибирск, «Топ-книга», тел. (3832) 36-10-26,
факс 36-10-27. Почтовый адрес: 630117, а/я 560.
E-mail: office@top-kniga.ru; http://www.top-kniga.ru

Тюмень, «Друг», ул. Республики, д. 211,
тел./факс (3452) 21-02-20.
E-mail: drug@tyumen.ru

Тюмень, «Фолиант», тел. (3452) 27-36-06, факс 27-36-11.
Почтовый адрес: 625039, ул. Харьковская, д. 83а.
E-mail: foliant@tyumen.ru

Татарстан

Казань, «Тайс», ул. Гвардейская, д. 9а, тел. (8432)
76-34-55, факс 38-24-32.
E-mail: tais@bancorp.ru

Урал

Екатеринбург, магазин № 14, ул. Челюскинцев, д. 23,
тел. (3432) 53-24-90.
E-mail: gvardia@mail.ur.ru

Екатеринбург, «Валео Плюс», тел. (3432) 42-07-75, 42-56-00.
E-mail: valeo@emts.ru

Пермь, «Азбука», тел. (4322) 41-01-41, факс 64-12-26.
Почтовый адрес: 614010, г. Пермь, ул. Героев Хасана, д. 10.
E-mail: azb_plus@perm.raid.ru

70,0
Y.P.

C++

Специальный справочник

В книге реализованы две глобальные цели. С одной стороны — это справочник, который содержит исчерпывающую информацию об основных элементах C++, а с другой — учебник, в котором последовательно рассказывается о принципах создания пользовательских программ.

Справочные материалы собраны в тематические разделы, которые помогут быстро найти элемент, необходимый для конкретной ситуации, а учебные материалы, опираясь на справочную информацию, предоставляют возможность осваивать язык активно — в процессе разбора программ-примеров.

Это уникальное сочетание справочника и учебного пособия, при, казалось бы, небольшом объеме книги, делает ее полезной не только начинающим программистам, но и пользователям, активно работающим с C++.

Уровень пользователя



Начинающий Опытный Профессионал

Категория книги: Справочник
языки программирования

Борис
Карпов

Татьяна
Баранова



**Информация,
которую вы найдете
в справочнике:**

- полный словарь терминов и понятий языка C++
- типы данных, операторы и ключевые слова
- особенности программирования на C++
- директивы, макросы и операторы препроцессора
- библиотечные функции
- отличия языков C и C++
- справочные таблицы и многое другое...



*На каждый вопрос —
готов ответ!*

ПИТЕР®
WWW.PITER-PRESS.RU

Посетите наш Web-магазин: <http://www.piter-press.ru>

Борис Карпов
Татьяна Баранова



Стенный журнал